Fernando Magno Quintão Pereira

Arcademis: Um Arcabouço para Construção de Sistemas de Objetos Distribuídos em Java

Dissertação de Mestrado apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

Dezembro de 2003

Resumo

Esta dissertação apresenta Arcademis: um arcabouço voltado para o desenvolvimento de sistemas de *middleware*. Os componentes que constituem este arcabouço formam um conjunto de classes abstratas, interfaces e componentes concretos, implementados em Java, que podem ser estendidos ou combinados de várias maneiras a fim de gerar plataformas adequadas a diferentes cenários. Com o intuito de ilustrar a utilização do arcabouço proposto, este é usado para instanciar uma plataforma de *middleware* adequada à versão J2ME da linguagem Java. Esta plataforma, denominada RME, fornece um serviço de invocação remota de métodos para o perfil CLDC de J2ME.

Abstract

This master dissertation presents Arcademis: a framework for middleware development. This framework is constituted by a set of Java classes and interfaces, plus some concrete components that can be extended or assembled together in order to build middleware platforms that support several different requirements and scenarios. A middleware system, obtained from Arcademis, is also presented. This platform, called RME, provides J2ME, the Java distribution for embedded and mobile devices, with a remote invocation service. This service targets CLDC, a particular configuration of J2ME that does not support the Java reflexivity features.

para os caras estranhos

Agradecimentos

Agradecer sempre é difícil: sempre se corre o risco de esquecer alguém. Assim, acho que primeiro eu vou agradecer a todo o mundo, que se eu acabar não citando alguma pessoa que me ajudou, seja direta ou indiretamente, ela não estará esquecida de todo. Mas creio que o mais justo é que eu me lembre dos professores do Laboratório de Linguagens primeiro: o Bigonha, a Mariza e o Marco Túlio, pois eles participaram diretamente do projeto de pesquisa que culminou nesta dissertação. Claro, eu não posso deixar de lembrar dos outros professores do DCC, pois se gosto de Computação, eles têm parte nisto. E fica também meu agradecimento aos irmãozinhos do Laboratório, pois eles propiciaram um excelente ambiente de trabalho.

Existe também um punhado de gente que não tem nada a ver com a dissertação, mas tem a ver comigo, então eu não poderia deixar de lembrar deles aqui. Meus pais, por exemplo. Eles não me deram idéias para a dissertação, tampouco corrigiram meus parágrafos, mas em compensação, que pais eu tenho! São minha riqueza; e me considero milionário. Além dos meus pais, eu tenho dois irmãos muito legais. Um deles inclusive, eu descobri que joga *Mega Man* melhor que eu.

Ah, convém também que eu não esqueça de meus amigos. Em particular, quando a Vanessa for famosa, poderei dizer que o nome dela aparece em minha dissertação. Além dela, eu tenho um punhado de amigos para lembrar: deixei vários em Nova Era, e encontrei outro tanto em BH. Gosto de todos, mas existem dois que são especiais: o Adriano e o Almir. Aprendi muito com eles; como também aprendi com a Aline: minha irmãzinha, que sempre emprestou alguma cor a estes dias cinzentos de Belo Horizonte.

Finalmente, eu preciso agradecer ao Wendell, ao Cristiano e ao Ademir: coisa combinada: é que nós, aprendizes de cientista, precisamos de citações :)

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Aplicações Distribuídas e Sistemas de *Middleware*

A demanda por aplicações distribuídas tem crescido continuamente desde o surgimento das mesmas. A Internet, por exemplo, experimentou na última década um crescimento vertiginoso, e atualmente é utilizada por dezenas de milhões de pessoas em diferentes países [Gei01]. Hoje, é possível pesquisar por diversos tipos de informações, trocar mensagens e, até mesmo, realizar compras ou transações bancárias nesta rede mundial. Sobre a Internet são executadas várias aplicações distribuídas diferentes. Algumas muito simples, outras extremamente complexas, como, por exemplo, os sistemas bancários. Existem também aplicações distribuídas que não utilizam a Internet. Estas são utilizadas em redes particulares de empresas, universidades ou órgãos públicos, o que contribui ainda mais para a difusão de programas distribuídos.

Aplicações distribuídas, em geral são executadas em ambientes heterogêneos, constituídos por computadores que, em muitos casos, apresentam arquiteturas e sistemas operacionais diferentes. Uma das soluções adotadas para amenizar os problemas decorrentes dessa heterogeneidade foi interpor entre aplicações e sistemas operacionais uma terceira camada de *software*. Esta terceira camada, denominada *middleware* [CK02], permite que desenvolvedores de aplicações distribuídas possam dispor de uma interface de programação uniforme. Assim como linguagens de programação permitem ao desenvolvedor abstrair-se de detalhes da arquitetura de máquina, plataformas de *middleware* permitem que o desenvolvedor de aplicações trabalhe de forma relativamente independente das primitivas de comunicação que um determinado sistema operacional disponibiliza.

Existem diferentes categorias de sistemas de *middleware*. As primeiras plataformas utilizavam técnicas procedurais, como troca de mensagens ou chamada remota de procedimentos, para prover comunicação entre aplicações distribuídas e baseavam-se no modelo cliente-servidor, segundo o qual uma aplicação cliente envia requisições para uma aplicação servidora responsável por tratá-las. Posteriormente surgiram plataformas de *middleware* orientadas por objetos, como Corba [Vin98, OMG99] e Java RMI [WRW96, Fra98]. Existem ainda plataformas de *mid-*

dleware baseadas em *espaços de tuplas*, como Lime [CVV01], PeerSpaces [VBBP02] e JavaSpaces [FHA99]. Vieram somar-se à tradicional arquitetura cliente-servidor outros modelos de comunicação, como a arquitetura *peer-to-peer* [RD01], por exemplo, aumentando ainda mais a diversidade de sistemas de *middleware* existentes. Embora muito variados, todos estes sistemas têm em comum o fato de encapsularem as primitivas básicas de comunicação fornecidas pelos sistemas operacionais, facilitando assim a tarefa de desenvolvimento de aplicações distribuídas.

Várias plataformas de *middleware* populares atualmente são sistemas monolíticos, isto é, são formados por um conjunto de componentes que, em muitos casos, não podem ser utilizados de forma independente [CK02]. Esta rigidez compromete a flexibilidade de tais sistemas, pois dificulta a utilização dos mesmos em alguns cenários específicos. Por exemplo, em computação móvel são usados dispositivos que geralmente possuem recursos muito limitados, como memória e capacidade de processamento. Sistemas como CORBA ou .NET, que são compostos por bibliotecas muito grandes, não podem ser integralmente executados na maior parte destes dispositivos. Além disto, tanto as implementações tradicionais de CORBA, quanto de .NET, bem como várias outras plataformas de *middleware*, não permitem que somente algumas de suas partes sejam utilizadas em um sistema distribuído a fim de diminuir o tamanho das bibliotecas e a quantidade de recursos necessários para suportá-las. Estas plataformas carecem de um projeto mais modular que lhes aumente o grau de flexibilidade.

Além de serem sistemas rígidos, as plataformas de *middleware* tradicionais não disponibilizam aos desenvolvedores de aplicações muitos parâmetros para reconfiguração. Por exemplo, o pacote Java RMI permite que diferentes algoritmos sejam aplicados sobre as mensagens que são transmitidas durante a invocação remota de métodos. Este mesmo pacote, contudo, não permite que a estrutura de um servidor seja modificada, por exemplo, para utilizar somente uma *thread* de controle em vez de várias, como normalmente acontece.

A fim de suprir a necessidade por sistemas de *middleware* mais reconfiguráveis, diversas plataformas já foram propostas desde a segunda metade da década de noventa até os presentes dias. Dentre estas soluções, citam-se, por exemplo, os sistemas TAO [SG97], dynamicTAO [RKC01, KRL+00] e UIC CORBA [RKC01], todos eles baseados na plataforma CORBA. Qualquer destes sistemas possui um projeto baseado em componentes, os quais podem ser facilmente modificados a fim de adequar a plataforma às necessidades do ambiente distribuído.

1.2 Objetivos da Dissertação

Seguindo uma linha de pesquisa semelhante à que deu origem aos sistemas reconfiguráveis introduzidos na Seção 1.1, esta dissertação define e implementa um arcabouço de componentes cujo objetivo é permitir o desenvolvimento de plataformas de *middleware* flexíveis e reconfiguráveis. Uma plataforma flexível permite que apenas as funcionalidades necessárias sejam utilizadas por uma aplicação. Uma plataforma reconfigurável, por sua vez, pode ser alterada de

diversos modos diferentes a fim de se adaptar às necessidades de algum cenário particular.

O arcabouço proposto, denominado **Arcademis** (*Arcabouço para o Desenvolvimento de Middlewares*), é constituído por um conjunto de interfaces e classes abstratas, implementadas na linguagem Java, e por uma coleção de classes concretas. Enquanto as classes abstratas podem ser implementadas de diferentes formas, de modo a atender às necessidades dos desenvolvedores de aplicações, as classes concretas podem ser facilmente combinadas a fim de gerar protótipos de plataformas de *middleware* rapidamente.

Além de implementar o arcabouço proposto, outro objetivo da dissertação é derivar dele um serviço de invocação remota de métodos adequado a KVM, a máquina virtual Java desenvolvida para aparelhos limitados computacionalmente [RTV01]. Esse serviço é útil para mostrar como as restrições que caracterizam aparelhos de comunicação sem fio podem ser atendidas por uma plataforma de *middleware* especialmente adaptada. Além disso, acredita-se que, dado ser este um serviço inédito, a sua implementação é por si só uma contribuição para a pesquisa relacionada a plataformas de *middleware*.

Finalmente, o estudo de padrões de projeto e sua aplicação no desenvolvimento de plataformas de *middleware* reconfiguráveis é outro objetivo desta pesquisa. Padrões de projeto são soluções comuns que atendem uma mesma família de problemas [GHJV94]. Por exemplo, quando for necessário garantir que exista somente uma instância de uma classe, pode-se recorrer a um padrão de projeto conhecido como *Singleton*. Alguns padrões tornam mais simples o desenvolvimento do *software*, enquanto outros facilitam sua reconfiguração.

1.3 Validação do Arcabouço Proposto

Existe uma carência de plataformas de *middleware* reconfiguráveis desenvolvidas para a linguagem Java. O principal sistema implementado para esta linguagem é conhecido como Java RMI (do inglês *Java Remote Method Invocation*) e fornece aos desenvolvedores de aplicações a possibilidade de invocar métodos sobre objetos distribuídos, isso é, que não estão localizados no espaço de endereçamento da mesma máquina virtual. Java RMI permite que aplicações distribuídas e não-distribuídas utilizem sintaxes muito parecidas, possibilitando que desenvolvedores obtenham todos os benefícios da programação orientada por objetos, ainda que tais objetos encontrem-se espalhados por uma rede de computadores.

Java RMI, contudo, possui limitações que o tornam inadequado para alguns cenários. Por exemplo, existe uma edição da linguagem Java conhecida como J2ME (*Java Micro Edition 2.0*) [RTV01] que foi desenvolvida para dispositivos limitados computacionalmente, como, por exemplo, aparelhos celulares. Dado que o universo de tais dispositivos é caracterizado por grande heterogeneidade, esta distribuição de Java possui diferentes configurações que atendem a famílias distintas de aparelhos. Há uma configuração, denominada CLDC (*Connected Limited Device Configuration*) voltada para a telefonia móvel que não suporta um mecanismo presente

em outras edições da linguagem conhecido como reflexividade. Reflexividade é a capacidade de um programa obter informações relativas à sua estrutura interna em tempo de execução. Java RMI utiliza reflexividade para serializar objetos e, portanto, não pode ser utilizado para o desenvolvimento de aplicações distribuídas em CLDC sem que sejam necessárias profundas modificações em sua estrutura.

Serialização de objetos, em Java, é o processo de converter objetos em seqüências de *bytes*, para transmiti-los por um canal de comunicação ou para gravá-los em meio permanente. Java RMI utiliza serialização para a transmissão de objetos entre clientes e servidores, e, em Java, a serialização necessita do mecanismo de reflexividade para reconstruir objetos a partir de cadeias de *bytes*. Caso Java RMI possuísse uma arquitetura mais flexível, então os algoritmos empregados para serializar objetos poderiam ser alterados de modo a não serem dependentes de reflexividade. Infelizmente, na implementação tradicional de Java RMI, tal modificação não é possível.

A fim de validar o arcabouço proposto, o mesmo foi utilizado para implementar um serviço de invocação remota de métodos em Java que não depende dos mecanismos de reflexividade desta linguagem. Tal serviço, denominado RME (*Remote Method Invocation for J2ME*) foi desenvolvido sobre o perfil CLDC e permite que aplicações distribuídas sejam desenvolvidas segundo um modelo muito semelhante ao utilizado por Java RMI. Testes realizados sobre KVM foram realizados a fim de comprovar a funcionalidade de tal serviço.

1.4 Os Três Níveis de Abstração

A fim de melhor entender como o arcabouço pode ser utilizado para facilitar o desenvolvimento de uma plataforma de *middleware*, um sistema distribuído implementado a partir dele pode ser dividido em três níveis, que, no contexto deste trabalho, são denominados *níveis de abstração*. O primeiro nível é formado pelas classes e interfaces que constituem o arcabouço. O segundo nível é definido por um conjunto de classes obtidas a partir das classes e interfaces existentes no primeiro nível, constituindo uma plataforma de *middleware* pronta. Finalmente, o último destes níveis é constituído pelas aplicações distribuídas que se comunicam usando as facilidades fornecidas pelo primeiro e segundo níveis.

O primeiro nível de abstração é formado pelos elementos que constituem Arcademis. Tratam-se de classes abstratas e interfaces, embora tal nível também contenha componentes concretos. Estes últimos podem ser combinados para compor plataformas de *middleware*, não sendo necessário que o desenvolvedor conheça detalhes acerca de sua implementação. Por outro lado, a fim de que o arcabouço possa ser utilizado de forma mais flexível, novas implementações para estes componentes podem ser fornecidas pelo desenvolvedor de aplicações. Existe, portanto, um compromisso entre a rapidez de desenvolvimento e a flexibilidade alcançada: enquanto a utilização de componentes concretos torna mais rápido o desenvolvimento do *middleware*, a im-

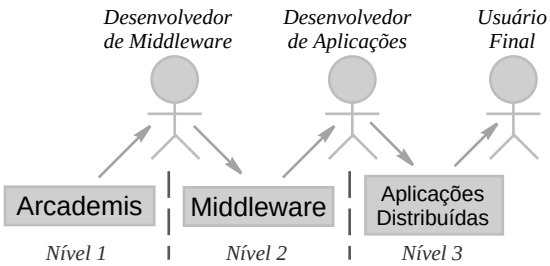

Figura 1.1: Os três níveis de programação em Arcademis.

plementação de componentes abstratos possibilita maior adequação a um conjunto de requisitos particulares.

O segundo nível que compõe essa estrutura de programação é constituído pela plataforma de *middleware* propriamente dita. Conforme anteriormente discutido, tal plataforma pode ser obtida de duas formas: via implementação de componentes abstratos ou por meio da utilização de componentes concretos definidos no primeiro nível de abstração. No segundo nível de abstração são decididas questões tais como o meio de comunicação empregado para a troca de mensagens e o protocolo de serialização usado para transformar objetos em seqüências de *bytes*. A separação entre a meta-programação e o desenvolvimento de aplicações dá-se nesse nível, ou seja, uma vez definido o conjunto de componentes que integra a plataforma de *middleware*, novas redefinições não podem ser feitas dinamicamente, ainda que tais modificações sejam teoricamente possíveis.

O terceiro nível representa a camada de aplicação em um sistema de objetos distribuídos. Por aplicação designa-se o grupo de programas construídos com o fim de prover serviços e abstrações para usuários que não são, necessariamente, programadores. O desenvolvimento destas aplicações é facilitado porque grande parte da complexidade inerente à programação distribuída está encapsulada pelos dois níveis de abstração descritos anteriormente. Existem diversos tipos diferentes de aplicações distribuídas, por exemplo: programas de leitura de correio eletrônico, livrarias virtuais e programas, como o MOSIX [BL98], capazes de coordenar *clusters* de computadores na realização de alguma atividade que demanda alto poder de processamento. A Figura 1.1 mostra a relação entre os três níveis de abstração e os usuários de cada um deles.

1.5 Observação Acerca de Termos Estrangeiros

Muitos dos termos estrangeiros cujo uso é considerado comum não são traduzidos, no presente texto, ao passo que outros são utilizados segundo a sintaxe da língua portuguesa. Por exemplo, o termo *middleware*, que em inglês, sua língua de origem, não possui plural, nesta dissertação pode ser encontrado em tal forma. Por considerar-se que a palavra *middleware* denota uma entidade concreta e enumerável (a camada de *software* interposta entre aplicações e o sistema operacional),

acredita-se que, quando utilizada na língua portuguesa, existe sentido em expressões tais como "Java RMI e CORBA são exemplos de *middlewares*".

Embora seja possível traduzir expressões tais como *skeleton* e *flyweight*, isto não é feito, uma vez que o uso de tais termos encontra-se consolidado como parte do jargão da área. Além disto, dificilmente se chegaria a um consenso quanto a tradução de expressões tais como *stub* ou *broker*, de modo que estas também não são traduzidas. Termos estrangeiros como *sotware* ou *stub* são grafados em itálico, ao passo que nomes de programas, tais como CORBA, e de linguagens de programação, por exemplo C++, são grafados normalmente.

1.6 Estrutura da Dissertação

Esta dissertação está organizada em cinco capítulos e um apêndice. No presente capítulo é feita uma rápida introdução sobre o tema a ser abordado. O segundo capítulo constitui uma revisão bibliográfica, onde são tratados os principais trabalhos e conceitos relacionados a plataformas de *middleware*, com ênfase aos sistemas baseados em invocação remota de métodos. No Capítulo 2 também são discutidos temas como arcabouços de desenvolvimento de *software* e padrões de projeto.

O Capítulo 3 trata da arquitetura de Arcademis e fornece alguns exemplos de reconfigurações que podem ser obtidas deste arcabouço. Neste capítulo são abordados os principais padrões de projeto utilizados na implementação de Arcademis. Além disto, diversos tipos de estruturas e características de plataformas de *middleware* são analisadas.

No Capítulo 4 é descrita a implementação de RME, a plataforma de *middleware* desenvolvida com o intuito de validar Arcademis, e que fornece para a edição J2ME da linguagem Java um serviço de invocação remota de métodos. Conforme é discutido naquele capítulo, *Java Micro Edition* apresenta duas configurações mais populares: CDC e CLDC, tendo sido RME desenvolvido para a segunda delas. Não existe na literatura sobre sistemas distribuídos outra implementação de um serviço de invocação remota de métodos para CLDC/J2ME.

No Capítulo 5 procura-se avaliar Arcademis em relação a critérios como generalidade e facilidade de uso. Além disso, tal capítulo compara o arcabouço proposto, bem como RME, com sistemas similares, buscando ressaltar os pontos positivos e negativos tanto de Arcademis quanto de sua instância. Conclusões e possíveis linhas de pesquisa que podem ser desenvolvidas a partir de Arcademis e de RME são apresentadas naquele capítulo.

Finalmente, exemplos de utilização de Arcademis são mostrados no Apêndice A. Tais exemplos incluem descrições de diferentes configurações que podem ser fornecidas para plataformas de *middleware* derivadas de RME. Alguns padrões de projeto utilizados na implementação de Arcademis são melhor descritos neste apêndice.

Capítulo 2

Revisão Bibliográfica

Neste capítulo são apresentados sistemas de *middleware* baseados em objetos distribuídos. Tal apresentação engloba desde sistemas precursores, como os Objetos de Rede introduzidos por Modula-3 no início dos anos 80, até o estado da arte neste campo: as plataformas de *middleware* reconfiguráveis estática e dinamicamente. Além disto, são abordados aspectos das arquiteturas de *middleware* mais populares, como CORBA, Java RMI e o modelo de objetos distribuídos de .NET. Além de discutir tais sistemas, este capítulo trata de arcabouços de desenvolvimento de *software*, de padrões de projeto e de *Java Micro Edition*, uma distribuição da linguagem Java adequada a dispositivos limitados em termos de capacidade computacional.

2.1 Visão Geral acerca de Sistemas de *Middleware* Baseados em Objetos Distribuídos

Plataformas de *middleware* orientadas por objetos baseiam-se no tradicional modelo cliente/servidor, em que objetos clientes solicitam a execução de operações por objetos servidores, sendo que estas duas partes podem não estar localizadas no mesmo espaço de endereçamento [MCE02]. Esse tipo de *middleware* disponibiliza aos desenvolvedores de aplicações distribuídas um alto nível de abstração, na medida em que torna transparente para eles a localização de objetos. Ao permitir que objetos distribuídos possam ser utilizados como se existissem todos no mesmo espaço de endereçamento, *middlewares* orientados por objetos disponibilizam aos desenvolvedores de aplicações uma poderosa abstração, pois estes podem se ater aos detalhes do modelo de objetos a ser desenvolvido sem, contudo, preocuparem-se com detalhes de comunicação inerentes a uma rede de computadores. Assim, programas distribuídos orientados por objetos são codificados segundo uma sintaxe muito próxima da que é utilizada em aplicações locais. Particularidades relacionadas ao tratamento de erros e exceções impedem que códigos de aplicações locais e distribuídas apresentem uma sintaxe exatamente igual [WWWK97].

Middlewares orientados por objetos evoluíram de um mecanismo conhecido como Chamada Remota de Procedimentos, ou RPC (do inglês *Remote Procedure Call*) [Ste98, CDK96]. Tal

técnica permite que um processo invoque procedimentos cuja implementação não se encontra disponível localmente. Sempre que uma rotina remota é invocada, seus argumentos são transformados em uma seqüência de *bytes* e transmitidos para a entidade responsável pelo processamento da chamada [WRW96]. A possibilidade de implementar a comunicação entre processos via rotinas de alto nível ao invés de primitivas de algum sistema operacional específico representou um grande salto de qualidade no projeto de sistemas distribuídos.

Em sistemas orientados por objetos, a chamada remota de procedimentos naturalmente cedeu lugar a invocação remota de métodos. Um dos primeiros sistemas de *middleware* orientados por objetos foi proposto para a linguagem de programação Modula-3 [Nel91]. Tal plataforma, conhecida como Sistema de Objetos de Rede (*Network Object System*) [BNOW93], introduziu muitos dos elementos atualmente presentes na maior parte dos *middlewares* orientados por objetos, tais como *stubs*, *skeletons*, *serialização* e *coleta de lixo distribuída*.

Um objeto de rede possui métodos que podem ser invocados por programas diferentes daquele onde o objeto foi criado. A entidade que invoca métodos remotos é conhecida como *cliente* e a entidade que processa as invocações é conhecido como *servidora*. Tanto o programa cliente quanto o servidor podem ser executados em computadores diferentes, ou em espaços de endereçamento diferentes no mesmo computador, ou ainda no mesmo espaço de endereçamento. Dado o nível de abstração fornecido pelo *middleware*, o processo cliente não precisa saber se um método é executado local ou remotamente.

Uma vez que os objetos de rede podem estar localizados em espaços de endereçamento diferentes, a comunicação entre eles se dá via entidades intermediárias, as quais são conhecidas como *surrogates* e *dispatchers*. *Surrogates* são os representantes do objeto remoto localizados no espaço de endereçamento de objetos clientes. Sua função é repassar para a implementação de um objeto as chamadas remotas destinadas a ele. Do lado do servidor, existem entidades denominadas *dispatchers*, que se encarregam de receber requisições de chamadas e repassá-las para o objeto responsável pelo seu processamento. Toda a transmissão de dados que caracteriza uma chamada remota é realizada entre *surrogates* e *dispatchers* de forma transparente para o usuário, que tem a ilusão de que os objetos de rede encontram-se todos no mesmo local. A Figura 2.1 torna mais clara essa relação entre cliente, servidor, *surrogate* e *dispatcher*.

Para o transporte de objetos, seja como parâmetro de uma invocação remota, seja como o valor de retorno da mesma, os objetos de rede utilizam uma técnica conhecida como serialização. Caso o objeto que precise ser transmitido não seja um objeto de rede, então ele é passado por valor, isto é, uma seqüência de *bytes* que representa uma cópia do objeto é transmitida como parâmetro ou valor de retorno. Em caso contrário, o objeto é passado por referência. Um objeto de rede é referenciado no espaço de endereçamento do cliente por uma estrutura denominada *wire reference*, que contém sua localização e seu identificador. Quando necessário transmitir um objeto remoto, sua referência (*wire ref*) é fornecida, em vez do próprio objeto.

O sistema adotado pelos objetos de rede de Modula-3, para a coleta de lixo distribuída,

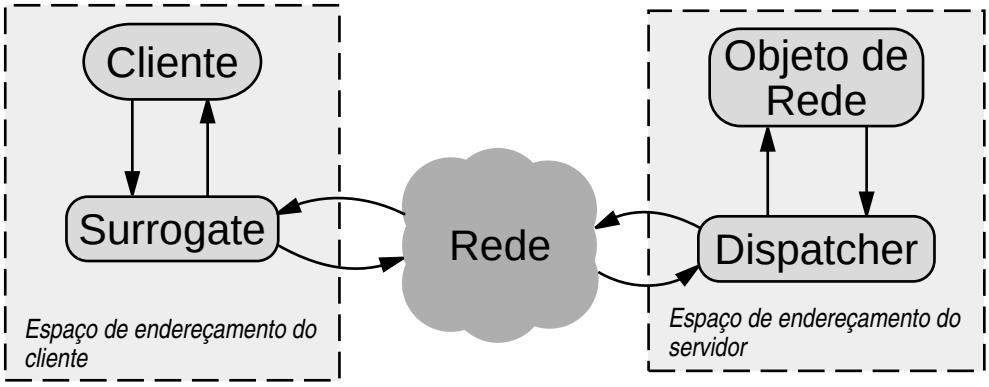

Figura 2.1: Objetos de rede em Modula-3.

baseia-se em contadores de referência. Para cada objeto distribuído é criada uma lista contendo os clientes que possuem *surrogates* para ele. Enquanto tal lista possuir elementos, um ponteiro para o objeto de rede é mantido ativo, o que impede que o coletor de lixo local remova tal objeto, ainda que não existam outras referências para ele. Tal procedimento é necessário, pois é possível que, em determinadas circunstâncias, existam somente referências remotas para um objeto. Quando a mencionada lista torna-se vazia, o ponteiro local para o objeto de rede é destruído, e ele torna-se passível de ser removido do ambiente de execução pelo coletor de lixo local.

Embora o coletor de lixo distribuído adotado pelos objetos de rede não seja capaz de lidar com ciclos de referências entre espaços de endereçamento diferentes, os algoritmos adotados são empregados em *middlewares* mais modernos, como Java RMI [Mic03], quase sem qualquer modificação. Além do coletor de lixo, outros conceitos introduzidos pelos objetos de rede fazem parte dos sistemas de *middleware* atuais. Dentre tais conceitos, talvez o mais importante seja a representação de objetos remotos via suas interfaces, por meio de *surrogates* (hoje conhecidos como *stubs*).

Mesmo sendo um sistema relativamente antigo, o modelo de objeto de redes de Modula-3 já previa algumas opções de reconfiguração. Por exemplo, o protocolo de transporte utilizado é implementado em um módulo separado do restante do sistema, de modo a poder ser facilmente alterado. Algumas possíveis opções de protocolo de transporte são TCP, UDP ou memória compartilhada. A plataforma também permite que o usuário especifique diferentes estratégias de serialização para tipos de dados particulares.

Os objetos de rede de Modula-3 foram os precursores de uma classe de *middlewares* atualmente muito popular entre os desenvolvedores de aplicações distribuídas. São exemplos de *middlewares* orientados por objetos as diversas implementações de CORBA [Ste93, Gro95, OMG99, Vin98], tais como Orbix [Bak97] e VisiBroker [NRV00]. Além de CORBA, existem outras plata-

formas orientadas por objetos, tais como Microsoft COM/DCOM [Rog97] e Java RMI [Mic03]. Existem também plataformas de *middleware* que utilizam largamente a invocação remota de métodos para permitir a comunicação entre aplicações distribuídas, como Enterprise JavaBeans [MH00].

2.2 Sistemas Atuais de *Middleware* Orientados por Objetos

A partir do surgimento das primeiras plataformas de *middleware*, iniciaram-se tentativas de padronização de arquiteturas distribuídas por parte de consórcios formados por grandes companhias. Resultou de tais esforços a especificação da arquitetura CORBA [Vin98, OMG99]. Outras plataformas de *middleware* que surgiram depois, patrocinadas por grandes companhias, são Java RMI [WRW96] e COM/DCOM [Fra98]. Embora estes três sistemas estejam baseados em um modelo orientado por objetos, eles possuem algumas diferenças, as quais são abordadas no restante desta seção.

2.2.1 CORBA

A arquitetura CORBA (do inglês *Common Object Request Broker Architecture*) [OMG99] foi desenvolvida nos primeiros anos da década de 90 e tem por objetivo permitir que componentes implementados por diferentes desenvolvedores e com diferentes tecnologias possam interagir entre si segundo um modelo de programação orientado por objetos. A plataforma CORBA pode ser separada em cinco componentes principais [Vin98]: o núcleo do sistema, a Linguagem para Definição de Interfaces, a Interface de Invocação Dinâmica, o Repositório de Interfaces e os objetos adaptadores.

O Núcleo do Sistema O núcleo da plataforma CORBA é denominado *Object Request Broker* (ORB). O principal propósito do ORB é entregar requisições de chamada de métodos para objetos e retornar para aplicações clientes os resultados de tais invocações. Os detalhes de implementação que o ORB utiliza para atingir tais objetivos são completamente transparentes para seus usuários. Uma vez que encapsula uma série de funcionalidades relacionadas à programação distribuída, o ORB permite que o desenvolvedor de aplicações possa abstrair-se de questões tais como a localização dos objetos, a maneira pela qual objetos são ativados ou como estes se comunicam.

Linguagem para Definição de Interfaces Um dos objetivos da plataforma CORBA é integrar aplicações implementadas em linguagens diferentes, permitindo que elas interajam segundo o mesmo protocolo. A fim de fornecer uma interface comum às diversas linguagens, CORBA propõe uma *linguagem para definição de interfaces*, ou IDL, a qual é utilizada para descrever os serviços disponibilizados por uma aplicação. Dessa forma, antes de uma aplicação poder utilizar

os métodos de um objeto localizado em um espaço de endereçamento diferente, ela deve conhecer a interface que o descreve. IDL fornece ao desenvolvedor tipos de dados básicos (`short`, `long`, `float`, `double` e `boolean`) e tipos compostos (`struct`, `union`, `sequence` e `string`). Para a descrição dos serviços fornecidos por um objeto, IDL fornece um tipo de dado denominado `interface`.

Repositório de Interfaces CORBA disponibiliza aos desenvolvedores de aplicações mecanismos de reflexividade via uma estrutura denominada Repositório de Interfaces, ou IR (do inglês *Interface Repository*). Esta estrutura armazena de forma persistente definições de interface, as quais são declaradas em IDL. Em CORBA, todas as interfaces estendem a interface `Object`, que define, entre outras operações, o método `get_interface()`. Tal método retorna uma referência para um objeto do tipo `InterfaceDef`, o qual descreve a interface do objeto em questão, cuja definição encontra-se armazenada no IR. A partir de uma definição de interface é possível conhecer sua cadeia de heranças e todas as operações que ela possui.

Interface de Invocação Dinâmica CORBA utiliza *stubs* e *skeletons* gerados a partir de interfaces definidas em IDL para permitir que operações remotas sejam invocadas. Neste caso, tais operações são invocadas com base em informações obtidas durante a compilação das interfaces IDL. Caso a aplicação não tenha acesso ao *stub* que representa um objeto remoto, ela pode utilizar um *stub* genérico para transmitir suas requisições para ele. Este componente, conhecido como *Dynamic Invocation Interface* (DII), é capaz de transmitir qualquer requisição para qualquer objeto remoto, sendo que a interpretação dos parâmetros de uma requisição, em tempo de execução, se dá por meio de consultas ao repositório de interfaces.

A fim de permitir que operações sejam invocadas dinamicamente sobre objetos cujo tipo pode não ser conhecido em tempo de execução, clientes podem criar requisições via o método `Object::createRequest`, que retorna um objeto do tipo `Request`. Um objeto do tipo `Request`, por sua vez, contêm operações como `addArg`, que adiciona argumentos à requisição que ele representa, e `invoke`, que efetivamente aciona a operação.

Objetos Adaptadores A plataforma CORBA lida com objetos que podem ter sido implementados via linguagens e técnicas muito diferentes. Por exemplo, interfaces IDL podem ser implementadas por um único programa, ou por vários *scripts* de algum sistema operacional. Devido a este fato, a plataforma CORBA deve ser flexível o suficiente para permitir que todos os tipos de objetos distribuídos possam usufruir da maior parte dos serviços que disponibiliza, tais como a própria invocação remota de métodos. Com tal fim, CORBA utiliza o conceito de objetos adaptadores (OA), que são, basicamente, as entidades responsáveis por integrar implementações de objetos remotos ao ORB.

Toda implementação de ORB baseada em CORBA deve fornecer um OA geral, conhecido como objeto adaptador básico, ou BOA (*Basic Object Adapter*). A estrutura e serviços providos

pelo BOA são flexíveis o suficiente para acomodar diferentes tipos de implementações de objetos, tais como, por exemplo, implementações compartilhadas em que múltiplos objetos coexistem no mesmo programa, ou, ao contrário, implementações em que cada método remoto é executado por um programa diferente.

Além de BOA, outros tipos de objetos adaptadores podem existir [PS98], embora os próprios projetistas de CORBA acreditem que uma quantidade pequena destes adaptadores é suficiente para cobrir as necessidades de todos os tipos possíveis de implementações de objetos [Vin98]. A título de exemplo, dois outros objetos adaptadores definidos pela especificação de CORBA são *portable object adapter* e *object oriented database adapter*. O primeiro deles permite que aplicações baseadas em CORBA sejam executadas sobre diferentes implementações de ORB [PS98], ao passo que o segundo adaptador fornece uma interface entre o ORB e sistemas de bancos de dados orientados por objetos.

2.2.2 COM/DCOM e .NET

O modelo de objetos distribuído de propriedade da Microsoft é conhecido como DCOM [Rog97]. Tal arquitetura transporta o modelo de objetos da Microsoft, conhecido como *Common Object Model* (COM), para o ambiente distribuído. Assim como CORBA, a plataforma DCOM permite que programas desenvolvidos em diferentes linguagens possam se comunicar. Com este fim, objetos distribuídos devem implementar uma ou mais interfaces que encapsulam tabelas de ponteiros para as entidades que realmente implementam os métodos remotos. Em DCOM, cada interface possui um identificador único, que também é usado para identificar o componente que a implementa.

A comunicação entre objetos, em DCOM, é feita via representantes locais, ou *proxies*, assim como em CORBA ou Java RMI. Porém, a nomenclatura utilizada em DCOM não é a mesma que a utilizada nestas duas outras plataformas: os representantes de objetos remotos do lado cliente de uma aplicação são chamados *proxies* e os representantes do lado servidor são chamados *stubs*. Tais componentes trocam mensagens por meio de um mecanismo de RPC implementado em sistemas operacionais da família Windows. Assim como CORBA, DCOM permite que chamadas remotas sejam executadas estática e dinamicamente. Chamadas estáticas são realizadas por meio de *proxies*, os quais são gerados automaticamente. Chamadas dinâmicas, por sua vez, são implementadas via bibliotecas de tipos que permitem a descoberta, durante a execução da aplicação, da assinatura dos métodos que fazem parte de uma determinada interface.

A portabilidade da arquitetura DCOM é penalizada pelo fato de vários dos serviços que ela fornece serem parte do sistema operacional Windows. A Microsoft tem procurado argumentar que esta plataforma é independente de qualquer sistema operacional, contudo, implementações robustas de DCOM existem somente para Windows [CK02].

O modelo DCOM serviu de base para o desenvolvimento da tecnologia .NET [OH01], que procura integrar os serviços tradicionalmente fornecidos pelos sistemas operacionais da Microsoft,

com aplicações implementadas em um vasto conjunto de linguagens de programação diferentes. Além de integrar programas implementados em diferentes linguagens, .NET também facilita o acesso de aplicações à Internet. Com este objetivo, a plataforma .NET utiliza um protocolo de comunicação baseado na linguagem de marcação XML [Mac03]. Tal protocolo, denominado SOAP [STK01], ou *Simple Object Access Protocol*, permite que os parâmetros e o resultado de uma chamada remota sejam conduzidos pela Internet por meio de mensagens codificadas em XML [Mac03].

A plataforma .NET permite a interação entre programas escritos em linguagens diferentes por meio de um ambiente de execução conhecido como *Common Language Runtime*, ou CLR. Tal arquitetura define um sistema de tipos normalizado, para o qual são fornecidos mapeamentos para diversas linguagens de programação. Dado um componente de *software* qualquer, este pode ser traduzido para CLR, desde que exista um mapeamento entre a linguagem na qual tal componente foi implementado e o sistema de tipos mencionado [CK02].

Nem toda linguagem de programação possui um mapeamento perfeito para o sistema de tipos de CLR. Assim, o mapeamento de alguns conceitos de linguagens não orientadas por objetos não é uma questão trivial. Suponha, por exemplo, que os métodos de uma interface remota sejam mapeados para programas separados implementados por alguma linguagem *script*, como Perl. Neste caso, não é simples garantir que todos os métodos compartilhem um estado comum, como seria possível caso eles fossem implementados pela mesma classe C++. Devido a este fato, nem todos os programas desenvolvidos sobre um sistema operacional da família Windows podem interagir via a plataforma .NET. Outra questão que compromete a portabilidade desta plataforma é o fato dela ser extremamente dependente de sistemas operacionais da Microsoft. Existem tentativas de desenvolver versões de .NET para outros sistemas operacionais, como por exemplo, o projeto Mono [HJS03], que visa criar uma instância aberta de .NET para o sistema Linux. Tal projeto, contudo, ainda encontra-se em estágio pouco desenvolvido.

Existe uma extensão de .NET voltada para a computação móvel conhecida como *.NET Compact Framework*. Assim como a versão .NET para redes fixas, este arcabouço permite que aplicações desenvolvidas em diferentes linguagens possam interagir em uma rede sem fio, desde que elas possuam um mapeamento para CLR. Entretanto, apenas versões para dispositivos embutidos das linguagens C# e Visual Basic foram mapeadas até o presente momento [WWB+03]. As bibliotecas que compõem o arcabouço compacto .NET ocupam um pouco menos que 2MB de memória, o que impede que elas sejam utilizadas em dispositivos muito limitados computacionalmente, como alguns modelos de telefones celulares.

2.2.3 Invocação Remota de Métodos em Java

Java RMI (*Java Remote Method Invocation*) [WRW96, Mic03] estende o modelo de objetos da linguagem Java, permitindo que objetos localizados em espaços de endereçamento de diferentes máquinas virtuais possam interagir entre si. Ao contrário de plataformas de *middleware* como

CORBA ou os objetos distribuídos de .NET, Java RMI não fornece suporte a programas escritos em outras linguagens diferentes de Java. Esta é uma desvantagem do sistema, pois o torna menos geral e dependente de uma linguagem de programação específica. Por outro lado, desta desvantagem também advêm pontos positivos. Em primeiro lugar, Java RMI pode aproveitar-se um sistema de tipos e um ambiente de execução de programas comum a toda a rede. Assim, em Java RMI não é necessário que o projetista de aplicações implemente uma interface entre a linguagem de desenvolvimento adotada e uma linguagem própria para definição de interfaces, como, por exemplo IDL de CORBA [OMG99].

Além disto, por utilizar a mesma plataforma em todos os nodos de um sistema distribuído, o modelo Java RMI apresenta ainda outras vantagens em relação aos sistemas mais genéricos. Em primeiro lugar, código executável, por exemplo: *stubs* e interfaces para objetos remotos, pode ser distribuído sob demanda. Devido a este fato, *bytecodes* de objetos distribuídos não precisam ser pré-instalados no espaço de endereçamento de aplicações clientes. Finalmente, a plataforma RMI também se beneficia dos mecanismos de segurança de Java. Estes mecanismos de segurança incluem a verificação automática de *bytecodes* e o controle do acesso à memória de dados e de instruções.

No sistema Java RMI, objetos reais podem ser passados como argumentos e como valores de retorno em invocações remotas de métodos. A fim de transportar objetos ao longo da rede, a plataforma RMI usa o mecanismo de serialização da linguagem Java para converter os grafos de referências que constituem tais objetos em seqüências de bytes. Segundo este princípio, qualquer tipo de dados serializável de Java, o que inclui tipos primitivos como inteiros e tipos compostos como objetos, pode ser passado como um parâmetro ou valor de retorno.

Java RMI é menos dependente de plataformas de execução específicas do que .NET, ou mesmo CORBA. Em relação a .NET, porque existem mais implementações de máquinas virtuais Java para diferentes sistemas operacionais do que implementações de CLR. E, em relação a CORBA, porque, uma vez que as implementações desta plataforma são codificadas em C++, elas precisam conter em seu código várias diretivas de pré-processamento, a fim de poderem ser compiladas em diferentes ambientes de execução, o que, em geral, não ocorre com programas escritos em Java.

A Arquitetura Básica de Java RMI

Em Java RMI objetos são utilizados como se estivessem todos localizados no mesmo espaço de endereçamento, muito embora este não seja sempre o caso. Assim, objetos remotos não são utilizados diretamente em uma aplicação, mas por meio de representantes locais denominados *stubs*, ou *proxies*. Um *stub* implementa a mesma interface remota que o objeto que representa, sendo a sua função enviar as requisições de chamadas de métodos para serem processadas pelo objeto remoto e receber os resultados de tal processamento.

Em Java RMI os parâmetros de uma invocação remota são passados por valor, isto é, quando

necessário transmitir um objeto, uma cópia do mesmo é enviada, ao invés de sua referência, como normalmente se dá em chamadas de métodos locais. A fim de permitir que objetos sejam copiados entre diferentes processos, Java RMI, assim como a grande maioria dos *middlewares* orientados por objetos, utiliza um mecanismo conhecido como serialização. Serializar um objeto consiste em convertê-lo em uma seqüência de *bytes*, a qual pode ser, posteriormente, armazenada em um arquivo binário ou transmitida via um barramento de rede. Do lado cliente de uma aplicação distribuída, o *stub* se encarrega de serializar os parâmetros de uma invocação e de restaurar o valor de retorno da mesma, quando este existir, para um objeto real.

Assim como em CORBA, em Java RMI existe uma entidade denominada *skeleton* que se encarrega de receber os parâmetros de uma invocação remota no lado servidor de uma aplicação. Este componente é utilizado como um objeto adaptador, pois permite o contato entre a implementação de um objeto remoto e a camada de transporte, por onde chegam chamadas de métodos provenientes de aplicações clientes. A relação entre *skeletons*, *stubs*, aplicações clientes e servidoras é mostrada na Figura 2.2. Nessa figura, os números indicam a ordem em que as mensagens são enviadas. Quando um método é invocado, as seguintes ações principais são realizadas:

1. o *stub* estabelece uma conexão com um servidor localizado no espaço de endereçamento do objeto remoto;
2. o *stub* serializa os parâmetros da requisição e os transmite para o servidor remoto;
3. o servidor remoto cria uma nova *thread* para receber e tratar a chamada remota;
4. os parâmetros da invocação remota são repassados ao *skeleton*;
5. o *skeleton* extrai os parâmetros da invocação e os repassa para a implementação do objeto remoto;
6. a implementação do objeto remoto processa a requisição e envia o resultado para o *skeleton*;
7. o *skeleton* retransmite o resultado da invocação remota para o *stub*.
8. o *stub* retorna para a aplicação o resultado obtido remotamente.

A fim de permitir a localização de objetos distribuídos, Java RMI disponibiliza às aplicações um serviço de localização de nomes. Este serviço é constituído por um diretório onde nomes de objetos estão associados a *stubs*. Um cliente pode realizar buscas em qualquer banco de nomes, desde que seja possível estabelecer uma conexão com o computador em que o servidor se encontra. A plataforma Java RMI não permite que mais de uma instância do diretório de nomes exista no mesmo *host*.

O acesso ao serviço de nomes se dá via a classe `Naming`. Esta classe possui seis métodos, dos quais os mais utilizados são `lookup` e `bind` (ou `rebind`). O método `bind` permite que

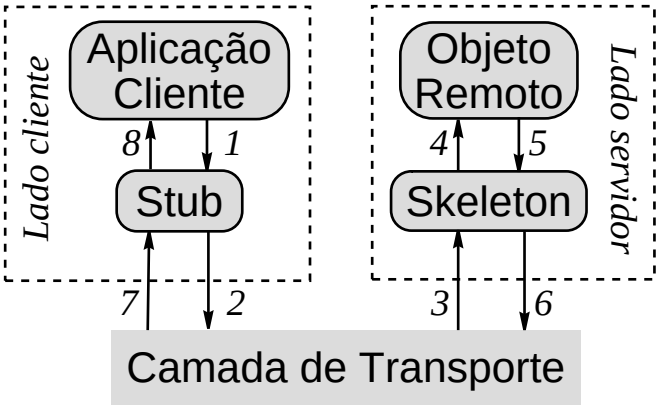

Figura 2.2: Invocação remota de métodos em Java RMI.

um objeto remoto registre-se junto ao servidor de nomes. Os argumentos deste método são o nome que identificará o objeto no diretório e o próprio objeto. Em RMI, sempre que um objeto remoto é passado como parâmetro de uma invocação remota, ao invés de ele próprio ser serializado e transmitido, o seu *stub* é transmitido no lugar. Como o método `bind` é remoto, ele envia ao diretório de nomes o *stub* de um objeto remoto, e não a própria implementação do mesmo. O segundo método da classe `Naming` citado é denominado `lookup`. Este método permite que aplicações clientes realizem buscas por nomes de objetos remotos. A Figura 2.3 mostra o relacionamento entre o servidor de nomes, objetos remotos e aplicações clientes.

2.2.4 A Arquitetura Monolítica de *middlewares* Tradicionais

Muitas plataformas de *middleware* são sistemas rígidos e pouco reconfiguráveis. Um sistema rígido não permite que algumas de suas partes sejam removidas caso não sejam necessárias em um determinado cenário. Um *middleware* pouco reconfigurável não disponibiliza aos seus usuários muitas opções para que suas características possam ser alteradas.

A rigidez, em conjunto com a baixa capacidade de reconfiguração, torna essas plataformas de *middleware* por demais inflexíveis. Por exemplo, implementações tradicionais de CORBA, como Orbix [Bak97], são constituídas por bibliotecas muito grandes, cujo tamanho é da ordem de várias dezenas de MB. O tamanho de tais distribuições as previne de serem executadas em dispositivos limitados, como telefones celulares. Por outro lado, algumas aplicações não necessitam de todas as funcionalidades fornecidas por tais sistemas. Uma aplicação cliente, em um *palmtop*, por exemplo, não precisaria dos componentes necessários ao recebimento de conexões. Entretanto, as partes desnecessárias de um *middleware* rígido não podem ser removidas; logo, estas plataformas não podem ser facilmente adaptadas para situações específicas.

O caráter rígido de algumas plataformas de *middleware* é responsável pela pequena capacidade de reconfiguração que as caracteriza. Segundo um princípio de programação orientada

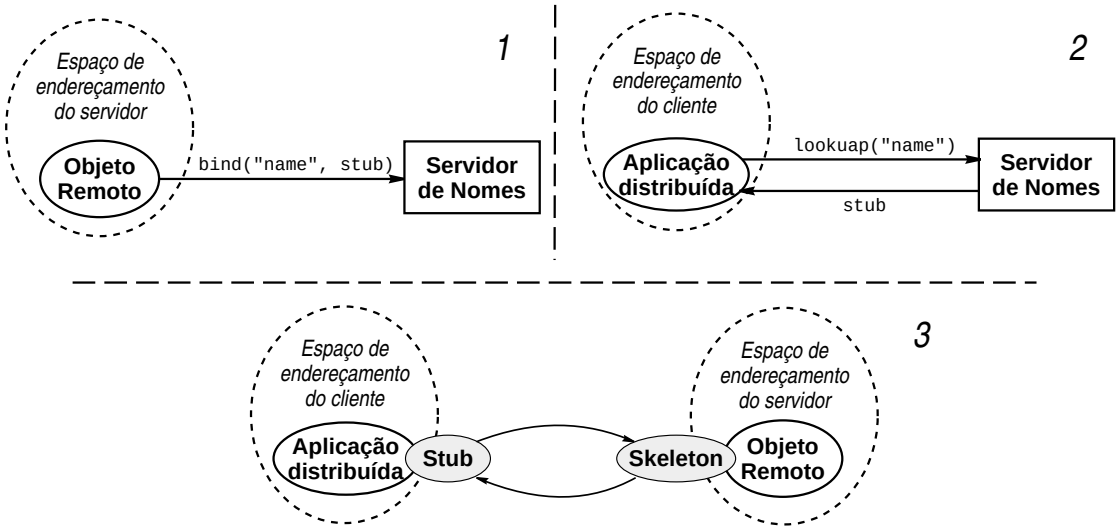

Figura 2.3: Relacionamento entre servidor de nomes, objeto remoto e aplicação cliente.

por objetos, denominado *princípio da abertura-fechamento*, quando necessário alterar um componente cujo desenvolvimento já esteja completo, recorre-se ao mecanismo de herança [Mey97]. Tal técnica, todavia, não pode ser facilmente empregada sobre os componentes de plataformas de *middleware* como CORBA, Java RMI ou .NET, uma vez que elas são constituídas por muitos pacotes fechados. O desenvolvedor de aplicações não tem como remover um de seus componentes e substituí-lo, sem alterar o código de outros elementos que compõem o sistema.

Por outro lado, estas mesmas plataformas não são sistemas completamente herméticos, pois disponibilizam alguns parâmetros que os usuários podem modificar. Por exemplo, o pacote Java RMI permite que diferentes algoritmos sejam aplicados sobre as mensagens que são transmitidas durante a invocação remota de métodos. Este *middleware*, contudo, não permite que a estrutura de um servidor seja modificada, por exemplo, para utilizar somente uma *thread* de controle em vez de várias.

Ainda em relação a Java RMI, nenhum dos componentes apresentados na Figura 2.2 pode ser facilmente alterado pelo desenvolvedor de aplicações de modo a tornar o pacote mais adequado a um certo conjunto de necessidades. Por exemplo, suponha que fosse necessário aumentar a tolerância a falhas de Java RMI. Com esse fim, o código de *stubs* poderia ser alterado para conter uma referência para um servidor principal, o qual deveria ser usado em situações normais, e uma outra referência para um servidor secundário, que seria utilizado apenas quando o primeiro deles não estivesse disponível. Esta alteração não pode ser realizada trivialmente no pacote Java RMI, entre outros motivos porque o usuário não tem acesso à parte da plataforma responsável pela efetuação de conexões entre aplicações clientes e servidoras.

Além de ser pouco reconfigurável, Java RMI também depende de alguns recursos da lingua-

gem Java que não estão disponíveis em todas as suas edições. Por exemplo, para a serialização de objetos, Java RMI utiliza os mecanismos de reflexividade da linguagem. Reflexividade permite que informações sobre a estrutura interna de um programa, por exemplo, o tipo dinâmico de um objeto, sejam conhecidas enquanto o mesmo está sendo executado. O perfil CLDC da versão J2ME da linguagem Java não possui a capacidade de reflexão computacional. Uma conseqüência desta última limitação é a impossibilidade de ser utilizado neste ambiente o pacote Java RMI, uma vez que tal sistema não pode ser reconfigurado de modo a utilizar uma estratégia diferente para a transmissão de objetos serializados.

Diferentes estudos têm sido realizados no sentido de aumentar a flexibilidade de sistemas tais como CORBA, Java RMI e .NET. Destes estudos resultou a implementação de algumas plataformas de *middleware*, sendo que muitas delas são voltadas para domínios específicos da computação, como a computação móvel ou sistemas de tempo real. No caso de CORBA, pode-se citar UIC CORBA [RKC01], LegORB [RMKC00] e minimumCORBA [Gro03]. Para .NET existe a implementação denominada *Compact Framework* [WWB+03] e existe uma implementação de Java RMI para o perfil CDC/J2ME, o qual caracteriza dispositivos dotados de mecanismos para o estabelecimento de conexões e algum poder de processamento, tais como alguns modelos de televisores e PDAs (*Personal Digital Assistant*). Afora estas implentações, existem outras linhas de pesquisa que visam desenvolver plataformas de *middleware* reconfiguráveis. Algumas dessas plataformas são descritas na próxima seção.

2.3 Sistemas de *Middleware* Reconfiguráveis

Nos últimos anos têm sido desenvolvidas plataformas de *middleware* que disponibilizam aos desenvolvedores de aplicações diversas opções de reconfiguração. Tais plataformas em geral apresentam um projeto modular, baseado em componentes cuja alteração não causa grande impacto sobre o sistema como um todo. Por exemplo, em um sistema de objetos distribuídos, não é necessário que o meio utilizado para o transporte de mensagens dependa do protocolo de serialização adotado. Assim, o fato de ser utilizado um canal de comunicação baseado em TCP ou UDP em nada deveria influir nos algoritmos de serialização utilizados pela plataforma de *middleware*.

Existem sistemas que apenas permitem que reconfigurações sejam feitas estaticamente, e existem sistemas, mais recentes, em que certos componentes podem ser alterados dinamicamente, isto é, durante a execução de aplicações. Por outro lado, existem sistemas que por si só não constituem plataformas de *middleware*, porém facilitam o seu desenvolvimento. Estes últimos são conhecidos como arcabouços, ou *frameworks*, e são abordados com maiores detalhes na Seção 2.5.

2.3.1 Plataformas Configuradas Estaticamente

O projeto de sistemas modulares, baseados em componentes relativamente independentes entre si, é uma conseqüência natural da programação orientada por objetos [Mey97]. A possibilidade de alterar partes específicas de uma plataforma de *middleware* enquanto preservando seus demais constituintes torna tais sistemas mais flexíveis, e, portanto, capazes de lidar com um conjunto maior de requisitos.

Outra vantagem da abordagem baseada em componentes é o fato de elementos não necessários em determinas condições poderem ser removidos da plataforma, de modo a torná-la menor. Por exemplo, todo sistema de *middleware* orientado por objetos apresenta funcionalidades exclusivas da parte cliente de uma aplicação ou da parte servidora da mesma. Existem situações onde uma aplicação comporta-se somente como cliente, ou somente como servidor de requisições. Nestes casos, os componentes não utilizados da plataforma deveriam poder ser removidos sem que o funcionamento da aplicação distribuída fosse comprometido.

Um dos primeiros sistemas de *middleware* baseado em componentes foi a plataforma TAO [SC99]. Este sistema utiliza arquivos contendo descrições de configurações para especificar as estratégias utilizadas, por exemplo, para tratamento de conexões, concorrência e escalonamento, as quais são definidas estaticamente durante a inicialização das aplicações. Com o intuito de aumentar o grau de reconfiguração da plataforma, seu desenvolvimento baseou-se em uma série de padrões de projeto [SC99]. Dentre tais padrões, citam-se, por exemplo, *reactor* [Sch94], um padrão que define um modelo de tratamento e notificação de eventos, *active-object* [SL95], um padrão que permite separar a invocação de métodos do processamento dos mesmos e *acceptor-connector* [Sch96], um padrão voltado ao estabelecimento de conexões.

TAO foi projetado para computadores embutidos em aviões, porém é adequado para o desenvolvimento de outros tipos de aplicações em que latência, determinismo e preservação de prioridades são requisitos essenciais [OSK+00]. A fim de poder atender a diferentes requerimentos, diversos elementos de TAO são passíveis de serem reconfigurados, por exemplo:

- o tipo de semântica adotada para o envio de mensagens [OSK+00]. As diferentes possibilidades são apresentadas na Seção 3.1.6;
- os algoritmos de serialização utilizados [SC99];
- as diferentes possibilidades de otimizar o envio de mensagens, de modo a poupar a utilização da rede [OSK+00]. Tais políticas são discutidas na Seção A.7;
- o número de *threads* responsáveis pelo processamento de invocações remotas[SL95];
- a estratégia de envio de mensagens, as quais podem ser despachadas síncrona ou assíncronamente [Sch96].

2.3.2 Plataformas Dinamicamente Reconfiguráveis

Existem aplicações distribuídas que executam em ambientes extremamente dinâmicos, isto é, que estão sujeitos a mudanças em suas características; mudanças estas muitas vezes imprevisíveis. Por exemplo, computadores móveis freqüentemente precisam lidar com flutuações na largura de banda disponível. Além disto, como existe uma grande variedade de dispositivos móveis, principalmente quando se considera aparelhos celulares, pode ser que determinada aplicação não se comporte do mesmo modo em diferentes máquinas. Aplicações que disponibilizam aos seus usuários uma interface gráfica, por exemplo, poderiam ser mostradas de um modo em um relógio de pulso e de outro em um *palmtop*.

Devido a estes fatos, existem fortes argumentos [RKC01, CK02] a favor de plataformas de *middleware* capazes de serem alteradas dinamicamente de modo a adequarem-se às novas características e propriedades do ambiente em que executam. De acordo com tais argumentos, diferentes tipos de aplicações, em diferentes contextos, podem ser desenvolvidos de modo a obter grandes benefícios da capacidade de alteração dinâmica de sistemas de *middleware*. Um exemplo típico em que a aplicação se beneficia da capacidade de reconfiguração dinâmica é um sistema de vídeo-conferência [CK02]. Neste cenário, uma aplicação que executa sobre um sistema de *middleware* dinamicamente reconfigurável pode selecionar o protocolo de transporte mais adequado em um dado momento, de acordo tanto com o nível de qualidade de serviço exigido por seus usuários quanto com a banda de transmissão disponível. Além disto, tal aplicação pode beneficiar-se do tipo de dispositivo em que está sendo executada. Assim, caso houvesse disponível um telão para a geração de imagens, a saída gerada pelo aplicativo poderia ser diferente daquela que seria produzida caso as imagens fossem geradas sobre a tela de um micro-computador.

Reflexão Computacional é o mecanismo normalmente utilizado para permitir que plataformas de *middleware* sejam modificadas dinamicamente. Conforme anteriormente dito, tal mecanismo permite que um programa tenha acesso a informações relacionadas à sua estrutura interna [KdRB91] em tempo de execução. Este caráter introspectivo da reflexão computacional permite a uma aplicação utilizar o conhecimento acerca do seu estado interno e do ambiente onde está sendo executada de modo a modificar seu comportamento quando diante de novas situações.

DynamicTAO

DynamicTAO [RKC01] é uma plataforma de *middleware* reflexiva implementada como uma extensão de TAO, sistema apresentado na Seção 2.3.1. Embora TAO seja uma plataforma portável e flexível, ela não permite reconfigurações dinâmicas, ou seja, uma vez que a plataforma é configurada, os algoritmos e componentes utilizados permanecerão os mesmos durante todo o tempo em que aplicações estiverem sendo executadas. DynamicTAO, por outro lado, suporta reconfigurações dinâmicas ao mesmo tempo em que assegura que o estado da plataforma e das aplicações distribuídas permanece consistente.

DynamicTAO é uma plataforma reflexiva porque é capaz de realizar auto-inspeção e auto-modificação em sua estrutura interna. Isso é possível porque o sistema mantém uma representação de seu próprio estado interno e de quais interações e dependências existem entre eles. Tal fato possibilita que componentes possam ser modificados sem que a execução de aplicações necessite ser reiniciada.

Os componentes que integram uma instância da plataforma TAO são mantidos agrupados em um repositório persistente de dados. Tais componentes são agrupados em categorias que representam os diferentes aspectos de um sistema de *middleware*, como, por exemplo, o protocolo de transporte ou o mecanismo de localização de objetos. Uma vez que um componente faz parte do repositório persistente, ele pode ser dinamicamente escolhido para fazer parte do *middleware*.

DynamicTAO fornece ao desenvolvedor de aplicações dois níveis diferentes de reconfiguração. No primeiro destes níveis, os componentes que podem ser alterados estão associados a entidades denominadas configuradores ou *component configurators*. Configuradores armazenam as dependências que existem entre os componentes e entre estes e a aplicação que os utiliza. Quando necessário alterar um dos componentes da plataforma de *middleware*, seu configurador é examinado, de modo a executar outras alterações a fim de que o sistema não seja deixado em um estado inconsistente.

O segundo nível de reconfiguração que caracteriza DynamicTAO utiliza componentes denominados *interceptadores* para que os desenvolvedores de aplicações tenham como alterar aspectos da plataforma de *middleware*. Interceptadores são elementos de *software* que podem ser inseridos em pontos particulares do *middleware*, por exemplo, entre o *stub* e a camada de transporte. Tais elementos podem ser utilizados com os mais diversos fins, por exemplo, para monitorar o comportamento de aplicações distribuídas, ou para compactar mensagens antes de transmiti-las.

UIC

UIC (*Universally Interoperable Core*) [RKC01], não é uma plataforma de *middleware* tal como TAO, ou mesmo DynamicTAO. Este sistema é, antes, uma infra-estrutura formada por diversos componentes abstratos interrelacionados. Componentes concretos, que podem ser combinados estaticamente, durante a compilação do sistema, ou dinamicamente, durante a execução de aplicações, definem as propriedades de uma plataforma de *middleware* particular. O projeto de UIC, baseado em componentes, permite que diferentes aspectos de plataformas de *middleware* possam ser configurados, como por exemplo: protocolo de transporte e serialização, forma de estabelecimento de conexões, semântica da invocação remota, política de prioridades, geração de referências remotas, registro e localização de objetos, interface do servidor, utilização de memória volátil e criação de *threads*.

Assim como DynamicTAO, UIC é um sistema que utiliza reflexividade para permitir a implementação de *middlewares* capazes de serem dinamicamente reconfigurados. Porém, enquanto DynamicTAO é uma arquitetura voltada para o desenvolvimento de aplicações de tempo real,

UIC destina-se ao ambiente heterogêneo e mutável que caracteriza a computação móvel. Dado seu objetivo principal, o suporte ao desenvolvimento de aplicações para computação móvel, UIC procura lidar com três questões principais: o conjunto heterogêneo de dispositivos, o ambiente dinâmico, e as limitações de aparelhos tais como telefones celulares e *palmtops*. Assim, com o intuito de poder ser utilizado em aparelhos que não dispõem de grande quantidade de espaço para armazenamento persistente, plataformas de *middleware* obtidas de UIC buscam fornecer a tais dispositivos somente um conjunto mínimo de funcionalidades capazes de atender às necessidades de suas aplicações.

Cada instância de *middleware* derivada de UIC é denominada uma *personalização* [1]. Uma personalização pode ser configurada como cliente, servidora ou *middleware* híbrido. A personalização cliente fornece às aplicações apenas as funcionalidades necessárias para realizar invocações remotas e receber as respostas. Da mesma forma, uma personalização servidora somente permite que aplicações processem chamadas remotas. Personalizações híbridas agrupam tanto capacidades de personalizações clientes quanto servidoras.

Uma instância de UIC pode ainda ser classificada como mono-personalizada, ou multi-personalizada. Uma plataforma mono-personalizada é capaz de interagir com somente um tipo de *middleware*, por exemplo, Java RMI ou CORBA, ao passo que uma plataforma multi-personalizada é capaz de interagir com vários sistemas diferentes. Contudo, um sistema multi-personalizado não é equivalente a uma coleção de sistemas simples. Dada uma instância multi-personalizada de UIC, esta é capaz de lidar com diversas plataformas de *middleware* sem a necessidade de interferências por parte da aplicação. Por outro lado, em uma coleção de instâncias mono-personalizadas, a aplicação precisa decidir explicitamente qual personalização utilizar.

Conforme anteriormente mencionado, UIC permite reconfigurações estáticas e dinâmicas. Plataformas definidas estaticamente não podem ser modificadas uma vez que a execução de aplicações tenha início. Se por um lado, tal fato restringe a flexibilidade do sistema, por outro, ele contribui para reduzir a complexidade da plataforma, e também o tamanho de suas bibliotecas, o que é um requisito essencial para sistemas desenvolvidos para dispositivos como telefones celulares, por exemplo. Testes realizados com uma personalização desenvolvida para CORBA mostraram que o projeto reconfigurável de UIC não compromete o desempenho de *middlewares* derivados deste sistema e também não faz com que suas bibliotecas sejam grandes a ponto de limitar o uso da plataforma. A título de exemplo, uma personalização cliente, projetada para ser executada em *palmtops* não ocupa mais do que 17KB [RKC01].

Considerações acerca de Reconfigurações Dinâmicas

Conforme dito anteriormente, existem diversos argumentos em favor de plataformas de *middleware* que possam ser reconfiguradas dinamicamente, porém, a utilização de reflexivade em tais sistemas ainda carece de suporte mais conclusivo. Em primeiro lugar, porque a própria neces-

[1] O nome original de tais instâncias é *personality*

sidade de reconfiguração dinâmica parece, em muitos casos, ser sobrevalorizada e, em segundo lugar, porque a incorporação de mecanismos de reflexividade às plataformas de *middleware* agrega considerável complexidade a tais sistemas. Além disto, é possível modificar partes de plataformas de *middleware*, em tempo de execução, sem que seja necessário recorrer à reflexividade computacional.

É fato que existem cenários em que aplicações se beneficiam de uma plataforma de *middleware* capaz de ser reconfigurada dinamicamente. Por outro lado, tais reconfigurações são casos específicos que podem muito bem ser tratados à parte, sem que toda a plataforma de *middleware* deva ser modificada de modo a permiti-las. Por exemplo, um algoritmo de transmissão de dados que se adapte à largura de banda disponível pode ser mais apropriado para a comunicação sem fio que a utilização de diversos protocolos independentes escolhidos de acordo com contingências impostas pelo ambiente. Além disto, o próprio usuário final pode cuidar de configurar a plataforma de *middleware* de acordo com suas necessidades. Por exemplo, ele pode ajustar a saída gráfica da aplicação que está utilizando de modo que esta se adeqüe ao tamanho da tela do dispositivo utilizado.

Além de não serem, em muitos casos, necessárias, as reconfigurações dinâmicas, em diversas situações, não são aplicáveis, pois elas podem deixar todo o sistema em um estado inconsistente caso venham a ser utilizadas. Assim, quando um componente de *middleware* é alterado durante a execução de uma aplicação, é necessário verificar se tal alteração não tem impacto sobre outros componentes locais, e sobre componentes correspondentes que são executados em outros elementos que integram o espaço distribuído. Por exemplo, suponha que, diante de uma súbita redução na banda de transmissão disponível, um *host* decida aplicar um algoritmo de compactação de dados sobre as mensagens transmitidas. Antes de utilizar tal algoritmo, é necessário que todas as partes envolvidas no processo de comunicação concordem em fazê-lo, e o façam de forma consistente, ou pode acontecer que a cooperação entre diferentes instâncias do mesmo *middleware* seja prejudicada porque enquanto uma delas utiliza mensagens compactadas, as outras não as transmitem desta forma.

Por fim, a capacidade de reconfiguração dinâmica baseada em reflexividade acrescenta um nível de complexidade ao sistema de *middleware* que não pode ser desconsiderado. Tal complexidade se reflete no tamanho das bibliotecas que integram a plataforma, pois passam a fazer parte das mesmas tanto o código necessário ao carregamento e remoção de componentes quanto todos os mecanismos responsáveis pelo monitoramento do estado interno do *middleware*.

2.4 Padrões de Projeto

Padrões de projeto começaram a fazer parte da metodologia de desenvolvimento de *software* a partir dos anos finais da década de 70. Um dos padrões mais conhecidos até hoje foi descrito naquela época, tendo sido denominado MVC (do inglês *Model View Controler*) [KP88]. Tal

padrão divide uma aplicação dotada de interface gráfica em três partes: a interface propriamente dita (*View*), as estruturas de dados (*Data Model*) e o controlador de eventos (*Controler*), que é a estrutura responsável por permitir a iteração entre os usuários finais da aplicação e sua interface.

Embora a utilização de padrões de projeto remonte ao início dos anos 80, foi somente durante os primeiros anos da década de 90 que evidenciou-se entre a comunidade acadêmica a necessidade de formalizar tais técnicas. Pertence a esta época a principal publicação na área, o livro *Design Patterns – Elements of Reusable Software* [GHJV94], cujos quatro autores: Gamma, Helm, Johnson e Vlissides passaram a ser conhecidos como "a gangue dos quatro". Esse livro apresenta 23 padrões de projeto empregados na programação orientada por objetos, discute os problemas de que tratam tais padrões e expõe as mais relevantes conseqüências decorrentes da utilização dos mesmos.

Em *Design Patterns*, os exemplos utilizados para ilustrar a utilização dos padrões foram apresentados em C++, embora alguns deles houvessem sido escritos na linguagem *Smalltalk*. Exemplos de tais padrões implementados na linguagem Java podem ser encontrados no livro *The Design Pattern Java Companion*. Os mesmos padrões também são descritos em *Smalltalk* [ABW98], porém com exemplos mais elaborados.

Na área de sistemas distribuídos e paralelos, destacam-se os trabalhos desenvolvidos por Douglas Schmidt na Universidade de Washington, St. Louis. Os mais importantes relatórios técnicos e artigos descrevendo padrões de projeto específicos para estruturas concorrentes ou distribuídas desenvolvidos por Schmidt e seus colaboradores podem ser encontrados no livro *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* [SSRB00]. Muitos dos padrões idealizados nos laboratórios da Universidade de Washington foram utilizados nesta dissertação para a implementação de Arcademis.

Uma sucinta compilação dos principais padrões de projeto utilizados no desenvolvimento de TAO, sistema de *middleware* descrito na Seção 2.3.1, é apresentada por Schmidt em [SC99]. Dentre os padrões descritos, citam-se *facade*, *reactor* [Sch94], *acceptor-connector* [Sch96], *active object* [SL95], *service configurator* [JS97], *strategy* e *abstract factory* [2].

A utilização de padrões de projeto no desenvolvimento de Arcademis se justifica, em primeiro lugar, porque eles facilitam o projeto de sistemas de *software* complexos, e em segundo lugar porque tais padrões permitem que diferentes aspectos de uma instância de Arcademis possam ser reconfigurados sem que grande impacto seja causado sobre os demais componentes que compõem o arcabouço. Por exemplo, dada a utilização de fábricas, a alteração de qualquer componente do sistema torna-se trivial: basta modificar a fábrica responsável pela sua criação.

[2] Optou-se por não traduzir estes nomes de padrões porque eles são comuns na literatura da área

2.5 Arcabouços para o Desenvolvimento de Sistemas Orientados por Objetos

Diversos autores, desde os primeiros anos da década de 80, têm procurado definir o conceito de arcabouços para o desenvolvimento de sistemas orientados por objetos. Embora as definições expostas por cada autor não sejam exatamente iguais, a maior parte delas tem em comum o fato de considerarem tais arcabouços como uma técnica de reutilização tanto de código fonte quanto de análise de projeto.

Ralph Johnson, um dos autores do livro *Design Pattern – Elements of Reusable Software* [GHJV94], no artigo entitulado *Frameworks, Components, Patterns* [Joh97], fornece duas definições para arcabouços, ou *frameworks*, como são normalmente conhecidos tais sistemas. Segundo a primeira delas, *frameworks* são um conjunto de classes abstratas e uma representação da maneira pela qual tais classes interagem. De acordo com a outra definição, *frameworks* são conceituados como o esqueleto de uma aplicação que pode ser modificado de acordo com a intenção de um desenvolvedor de programas. De acordo com Johnson, estas não são definições conflitantes pois, enquanto a última delas trata do propósito de um arcabouço, a primeira descreve a estrutura do mesmo.

Definições mais antigas de *frameworks* podem ser encontradas na literatura. O próprio Ralph Johnson, em um trabalho anterior [JF88], separa tais arcabouços em sistemas de caixa branca e caixa preta e os define como um formato de projeto para um grupo particular de aplicações mais um conjunto de classes total ou parcialmente implementadas.

Uma definição semelhante a adotada por Johnson foi dada por Peter Deutsch, também em fins da década de 80 [Deu89]. Segundo esta definição, *frameworks* são uma coleção de classes abstratas e os algoritmos a elas associados. Desenvolvedores de aplicações poderiam utilizar tais sistemas fornecendo implementações adequadas para os métodos não definidos nas classes abstratas. Assim, além de considerar a estrutura dos *frameworks*, em sua definição, Deutch leva em consideração também o próposito dos mesmos, ou seja, de que forma e com que objetivo poderiam ser utilizados.

A definição de *frameworks* e a maneira pela qual tais sistemas se relacionam com plataformas de *middleware* são abordados por Douglas Schmidt em diversos trabalhos relacionados [SSRB00, SC99, D. 99, D. 02]. Em [SB03], Schmidt procura expor como padrões de projeto, arcabouços e plataformas de *middleware* podem ser utilizados de forma cooperante. De acordo com tal exposição, *frameworks* são projetados com base em determinados grupos de padrões de projeto. Os sistemas de *middleware*, por seu turno, podem ser desenvolvidos de forma rápida a partir de arcabouços de objetos.

2.5.1 Sistemas "Caixa preta" e "Caixa branca"

Arcabouços para o desenvolvimento de programas podem ser classificados em sistemas do tipo "caixa-preta" e sistemas do tipo "caixa-branca" [Joh97]. Esta classificação diz respeito ao nível de visibilidade que caracteriza a implementação dos componentes do arcabouço. Um sistema do tipo caixa-branca expõe aos seus usuários aspectos internos de seus componentes. Tal não acontece com os sistemas caixa-preta.

O principal mecanismo de utilização de um arcabouço do tipo caixa-branca é a herança. Neste caso, alguns atributos da superclasse devem ser visíveis para a subclasse. Já arcabouços do tipo caixa-preta são utilizados mediante a composição de objetos. É importante ressaltar que a realização de uma interface não caracteriza um caso de arcabouço caixa-branca, pois, uma vez que esta não possui dados internos, não são quebradas restrições de visibilidade durante a sua implementação. Mesmo a implementação de classes, sejam elas concretas ou abstratas, pode não constituir um caso de uso de arcabouços caixa-branca, pois pode ser que a subclasse não tenha acesso aos atributos internos da superclasse.

Uma vez que arcabouços do tipo caixa-branca expõem para o usuário aspectos internos de seus componentes, eles são mais flexíveis: o desenvolvedor é capaz de modificar a implementação dos elementos que constituem este tipo de arcabouço de acordo com as suas necessidades. Por outro lado, arcabouços deste tipo exigem maior familiaridade por parte de seus usuários, os quais, devido à maior liberdade que possuem, podem alterar partes de componentes de modo inconsistente. Os arcabouços do tipo caixa-preta, por sua vez, são de utilização mais simples e segura; contudo, não são tão flexíveis quanto os sistemas caixa-branca.

2.5.2 Quarterware

O sistema conhecido como Quarterware [SSC98, Sin99] é um exemplo de arcabouço direcionado ao desenvolvimento de plataformas de *middleware* baseados em objetos distribuídos. Este sistema permite que sejam desenvolvidos diferentes tipos de *middleware* a partir de um conjunto de componentes de *software* pré-definidos. Quarterware, que foi implementado na Universidade de Illinois, em Urbana Champaign, é composto por um núcleo básico de componentes que podem ser alterados pelo desenvolvedor de *middlewares*, e por um conjunto de componentes auxiliares, que podem ser utilizados para agregar funcionalidades não essenciais ao núcleo principal.

O objetivo central de Quarterware é facilitar o desenvolvimento de sistemas de *middleware*. A abordagem proposta por esta arquitetura possui algumas vantagens em relação à abordagem tradicional, na qual plataformas são construídas sem o auxílio de componentes já prontos. Em primeiro lugar, Quarterware implementa algumas propriedades que são comuns a todos os sistemas de *middleware*, tornando, assim, mais rápido o desenvolvimento de tais plataformas. Além disto, os componentes que formam o arcabouço podem ser alterados a fim de satisfazerem um determinado conjunto de requisitos. Finalmente, Quarterware possui algumas características que facilitam a otimização de desempenho dos sistemas dele derivados.

Quarterware define alguns parâmetros principais a partir dos quais sistemas de *middleware* podem ser configurados. Diferentes instâncias de *middlewares* são obtidas a partir de diferentes implementações de tais parâmetros, os quais podem ser classificados em seis grupos distintos:

Serialização: a política de serialização depende do tipo de dado que está sendo serializado. Tipos primitivos podem ser serializados por rotinas pré-definidas, ao passo que tipos compostos podem implementar as próprias rotinas de serialização;

Referências remotas: referências remotas contêm as informações necessárias para que aplicações clientes possam se conectar às aplicações servidoras. Essencialmente uma referência remota contém a localização de um objeto; assim, dependendo da forma em que endereços são especificados no sistema distribuído, diferentes possibilidades para a implementação de referências remotas existem. Por exemplo, em Java RMI, referências remotas encapsulam o par formado por um endereço IP e por um número de porta. Em SOAP, uma referência remota pode especificar a localização de um componente em um sistema de arquivos [GSI03], sendo, assim, constituída por uma URL (*Universal Resource Locator*) seguida por um caminho em uma hierarquia de diretórios. Já em aplicações paralelas que executam no mesmo computador, pode-se definir referências remotas como componentes que encapsulam um endereço de memória compartilhada;

Transporte: os mecanismos de transporte permitem a transmissão de dados. Diversos protocolos podem ser utilizados com tal fim. Java RMI, por exemplo, utiliza normalmente o protocolo TCP/IP, porém, quando necessário lidar com mecanismos de segurança impostos por *firewalls*, esta plataforma pode utilizar o protocolo HTTP.

Despacho: o despacho de requisições demanda a localização do servidor, algum pré-processamento da requisição, e a transmissão dos dados que constituem a chamada. Cada um destes passos admite variações. Por exemplo, objetos remotos podem ser localizados via um serviço centralizado de nomes, ou então podem ser localizados via um algoritmo semelhante à busca em grafos, em que cada nodo do grafo é constituído por um *host* que integra a rede;

Política de invocação: requisições remotas podem ser processadas segundo diferentes políticas, por exemplo, com uma *thread* para cada requisição, ou com uma *thread* para cada cliente, ou ainda com somente uma *thread* para processar todas as requisições.

Protocolo de comunicação: o protocolo de comunicação é definido pelo conjunto de mensagens que são trocadas durante as iterações entre objetos distribuídos, e por uma máquina de estados que define em que contextos tais mensagens são produzidas e enviadas. A título de exemplo, CORBA utiliza um protocolo denominado GIOP (*General Inter-Orb Protocol*), enquanto Java RMI utiliza JRMP (*Java Remote Method Protocol*).

Quarterware é um sistema muito mais flexível que os descritos nas Seções 2.3.1 e 2.3.2, pois, ao contrário daqueles, não é uma plataforma de *middleware*, mas um arcabouço a partir do qual outras plataformas podem ser desenvolvidas. Assim, Quarterware não se limita ao desenvolvimento de plataformas de *middleware* orientadas por objetos, embora este seja o seu objetivo principal. Por exemplo, existem implementações de CORBA, RMI e MPI baseadas em Quarterware, sendo que MPI [MPI95] (do inglês *Message Passing Interface*) é um sistema baseado em troca de mensagens.

Exemplos de Uso de Quarterware

Quarterware é constituído por um conjunto de componentes implementados em C++ cujo projeto foi, em grande parte, influenciado por CORBA. Devido a este fato, o desenvolvimento de uma instância desta plataforma a partir de Quarterware não é uma tarefa extremamente difícil. Tal instância foi desenvolvida como um dos primeiros exemplos de uso de Quarterware e mostrou-se tanto eficiente em termos de desempenho quanto em termos de tamanho, sendo constituída por bibliotecas relativamente pequenas se comparadas com as implementações tradicionais de CORBA. Além destes benefícios, a versão de CORBA obtida de Quarterware pode ser reconfigurada de diferentes modos. Por exemplo, qualquer dos protocolos: TCP ou UDP, pode ser utilizado para o transporte de mensagens, e diferentes políticas de utilização de *threads* podem ser adotadas.

Outra instância de *middleware* derivada de Quarterware foi uma versão de Java RMI implementada em C++. Nesta versão foram implementados os protocolos de serialização e comunicação (JRMP) utilizados por RMI. Dado que os componentes que constituem Quarterware são implementados em C++, algumas características de RMI, que é implementado em Java, não puderam ser fornecidas integralmente. Por exemplo, apenas objetos cujo tipo é conhecido em tempo de compilação podem ser serializados. Além disto, como em Quarterware objetos Java são representados por `structs` implementadas em C++, métodos não podem ser invocados sobre eles. Assim, caso um cliente RMI implementado em C++ receba como valor de retorno de uma chamada um objeto Java, os métodos deste objeto não podem ser invocados, embora o valor de seus atributos possa ser lido.

Finalmente, também uma instância de MPI foi derivada de Quarterware, embora este sistema não seja um exemplo de *middleware* em que o processo de comunicação se baseia em invocação remota de métodos. MPI, originalmente projetado para coordenar aplicações concorrentes, utiliza troca de mensagens para permitir a comunicação entre processos. Ainda assim, a instanciação deste sistema foi relativamente simples, sendo que diversas otimizações puderam ser implementadas sobre a instância obtida. Um exemplo de otimização foi a utilização de memória compartilhada para prover comunicação entre processos localizados na mesma máquina, ao invés de conexões TCP/IP.

Quarterware permite que seus componentes sejam otimizados para garantir bom desempenho

na comunicação entre processos ou objetos distribuídos. Assim, todas as três instâncias de *middleware* descritas anteriormente superaram em termos de latência as implementações dos sistemas originais. A latência é o tempo que uma tarefa demora para ser realizada por um sistema computacional. A título de exemplo, a latência em Visibroker, uma implementação de CORBA, foi medida como 1.5 vezes maior que a latência do sistema correspondente derivado de Quarterware.

Diferenças entre Arcademis e Quarterware

A principal diferença entre Arcademis e Quarterware diz respeito à arquitetura de tais sistemas. Os componentes implementados para os dois arcabouços são fundamentalmente diferentes, como também são diferentes os parâmetros de configuração enumerados em cada um destes sistemas. Enquanto Quarterware descreve reconfigurações sobre os seis aspectos descritos no início desta seção, Arcademis define 12 diferentes aspectos reconfiguráveis de plataformas de *middleware*, conforme é descrito na Seção 3.1. Assim, em Arcademis, o desenvolvedor de *middleware* dispõem de maior flexibilidade para configurar seus sistemas.

Quarterware é um sistema mais geral que Arcademis, pois não é um sistema dependente de uma linguagem de programação específica. Uma vez que Quarterware é um sistema baseado em CORBA, é possível derivar deste arcabouço plataformas de *middleware* que suportam aplicações implementadas em diferentes linguagens. A generalidade de Quarterware, contudo, tem o seu preço, na medida em que impede que suas instâncias se beneficiem de características específicas de determinadas linguagens de programação. Por exemplo, a implementação de Java RMI obtida de Quarterware não tira proveito dos recursos de polimorfismo que a linguagem Java possui. Quarterware está implementado em C++, e as funcionalidades que tal linguagem apresenta para a descrição de tipos em tempo de execução são muito limitadas se comparadas com as fornecidas por Java [Sin99]. Por exemplo, na implementação de RMI derivada de Quarterware, é possível serializar somente objetos cujo tipo é conhecido em tempo de compilação.

Em Java tal problema não existe, pois a linguagem dispõe de um mecanismo para carregar classes dinamicamente (*class loader*), de forma que, caso o tipo de um objeto não seja conhecido, a classe que o representa pode ser copiada para o espaço de endereçamento local. Instâncias de Arcademis podem utilizar o carregamento dinâmico de classes para resolver tipos desconhecidos, embora nem todas as edições da linguagem Java disponham desta funcionalidade. Por exemplo, o perfil CLDC de J2ME não fornece ao desenvolvedor de aplicações a capacidade de carregar classes dinamicamente.

A fim de melhor ilustrar esta limitação de Quarterware, suponha que duas classes façam parte de uma aplicação: `Classe` e `SubClasse`, sendo que a segunda delas estende a primeira. Métodos que retornam objetos do tipo `Classe` podem retornar também objetos do tipo `SubClasse`, porém, durante a execução, `SubClasse` pode não estar presente na lista de classes conhecidas por uma máquina virtual cliente. Em Java, caso fosse necessário reaver um objeto do tipo `SubClasse`

de uma seqüência serializada de *bytes*, o carregador de classes poderia ser utilizado para obter as informações necessárias a respeito deste componente, ao passo que em C++ tal opção não é possível.

Ambos os arcabouços também diferem quanto à implementação. Quarterware está implementado em C++ e os componentes de Arcademis estão implementados em Java. Finalmente, a última diferença entre os dois arcabouços é o momento em que reconfigurações podem acontecer. Arcademis prevê que plataformas de *middleware* somente sejam configuradas estaticamente, ao passo que Quarterware contém estruturas que visam possibilitar reconfigurações dinâmicas, embora nos exemplos de uso deste sistema tais estruturas não tenham sido utilizadas [SSC98, Sin99].

2.6 A Plataforma J2ME

A linguagem Java possui três distribuições principais, ou edições. Dentre estas edições, a mais popular é conhecida como J2SE, ou *Java 2 Standard Edition*. A segunda distribuição da linguagem, denominada J2EE [SHM+00], ou *Java 2 Enterprise Edition*, foi desenvolvida para atender aplicações que demandam grande robustez e segurança, sendo muitas vezes executadas em servidores de grande capacidade, como, por exemplo, aqueles voltados para serviços de comércio eletrônico. Finalmente, para o mercado constituído por dispositivos de menor capacidade computacional existe a plataforma J2ME, ou *Java 2 Micro Edition*. Cada uma destas edições define um conjunto de tecnologias que podem ser utilizadas para o desenvolvimento de aplicações. São partes de uma plataforma Java a especificação de uma Máquina Virtual, um conjunto de classes relacionadas e as ferramentas necessárias à instalação e configuração de aplicações.

A plataforma J2ME veio suprir as necessidades de um mercado cada vez maior de dispositivos computacionais, que vão desde *pagers* e *palmtops*, até aparelhos televisores com acesso à Internet. Assim sendo, um dos principais objetivos da plataforma J2ME é definir soluções que sejam válidas para todas estas tecnologias e padrões. Com o intuito de classificar e padronizar a enorme variedade de dispositivos existentes no mercado, foram definidos para o ambiente J2ME os conceitos de *configurações* e de *perfis*.

Uma configuração define uma plataforma mínima para uma determinada categoria de dispositivos móveis. Tais categorias contêm aparelhos com características similares quanto à capacidade de processamento e à memória disponível. Em termos mais concretos, uma configuração define uma máquina virtual, um conjunto mínimo de bibliotecas e os recursos da linguagem Java que estão disponíveis para os dispositivos de uma determinada categoria. A plataforma J2ME possui atualmente duas configurações principais. A primeira configuração, denominada CLDC (*Connected, Limited Device Configuration*), destina-se a dispositivos dotados de mecanismos de transmissão de dados e que possuem entre 160KB e 500KB de memória disponível. Exemplos típicos incluem os telefones celulares digitais e os assistentes pessoais digitais (PDAs). A se-

gunda destas configurações, denominada CDC (*Connected Device Configuration*), está voltada para dispositivos dotados de maior capacidade computacional, com no mínimo 2 MB de memória disponível [RTV01].

Um perfil atende às demandas específicas de uma certa família de dispositivos. Enquanto uma configuração visa aparelhos que possuem recursos de *hardware* semelhantes, um perfil é definido para dispositivos que executam tarefas semelhantes. Ao contrário de uma configuração, perfis incluem bibliotecas mais específicas, e vários deles podem ser suportados pelo mesmo dispositivo. Além disso, as classes que fazem parte de um perfil tipicamente estendem aquelas definidas para uma configuração. MIDP, ou *Mobile Information Device Profile*, foi o primeiro perfil definido para a plataforma J2ME, tendo sido lançado em novembro de 1999. Este perfil foi implementado sobre a configuração CLDC. Também para a configuração CDC foi desenvolvido um perfil, o qual ficou conhecido como *Foundation Profile* [RTV01].

A fim de fornecer à plataforma J2ME um ambiente no qual aplicações pudessem ser executadas, foi desenvolvida uma máquina virtual específica, denominada KVM. Esta máquina virtual foi implementada para executar em dispositivos dotados de um processador de 16 ou 32 bits e que não dispõem de mais que algumas centenas de *kilobytes* de memória. Essas especificações aplicam-se a uma vasta gama de aparelhos, como por exemplo telefones celulares digitais e *pagers*, dispositivos de áudio e vídeo portáteis e também pequenos terminais de consulta ou pagamento de débitos.

Tendo sido projetado para dispositivos limitados em termos de capacidade computacional, o perfil CLDC possui diversas restrições que não estão presentes em outras versões da linguagem Java. Dentre estas limitações cita-se a não existência do tipo ponto flutuante, utilizado para representar números reais. Outra limitação importante é a ausência de suporte aos mecanismos de reflexividade presentes na linguagem Java, os quais permitem que informações sobre a estrutura interna de um programa, por exemplo, o tipo dinâmico de um objeto, sejam conhecidas enquanto o mesmo está sendo executado.

Uma conseqüência desta última limitação é a impossibilidade de utilizar neste ambiente o pacote Java RMI, pois este faz uso de reflexividade para passar objetos serializados como parâmetros de invocações remotas ou como valores de retorno das mesmas [Mic03]. Caso esta plataforma possuísse uma arquitetura mais reconfigurável, o mecanismo de serialização poderia ser modificado de modo a não depender de reflexividade. Infelizmente esse tipo de alteração não é possível. Assim, um dos objetivos desta pesquisa é desenvolver sistemas de *middleware* que permitam alterações tão profundas quanto a técnica de serialização utilizada.

2.7 Conclusão

Este capítulo apresentou os conceitos de plataformas de *middleware*, *frameworks* e padrões de projeto e mostrou como tais conceitos se relacionam a Arcademis, um arcabouço para o de-

senvolvimento de *middlewares* orientados por objetos cuja implementação se baseia em muitos padrões de projeto conhecidos. Neste capítulo também foi apresentada uma visão geral acerca de sistemas de *middleware* orientados por objetos e das plataformas mais conhecidas deste grupo, a saber: CORBA, .NET e Java RMI. Embora estes sistemas sejam largamente utilizados pela indústria de *software*, existem muitos cenários onde os mesmos não podem ser empregados, uma vez que apresentam uma arquitetura monolítica que dificulta adaptações. A seguir, apresentou-se plataformas de *middleware* reconfiguráveis, como TAO e UIC CORBA, as quais foram desenvolvidas para sanar problemas advindos do caráter pouco modular de *middlewares* tradicionais. São duas as principais diferenças entre tais sistemas e Arcademis. Em primeiro lugar, destaca-se o fato do arcabouço proposto ser implementado em Java, ao passo que os sistemas discutidos nesta seção são implementados em C++. Além disto, Arcademis não é uma plataforma de *middleware* pronta para ser usada, mas um arcabouço a partir do qual tais sistemas podem ser obtidos. A discussão acerca de *middlewares* orientados por objetos encerrou-se com a apresentação de Quarterware, um arcabouço desenvolvidos com os mesmos objetos de Arcademis, porém implementado em C++. Assim, embora já existam plataformas de *middleware* reconfiguráveis e arcabouços para o desenvolvimento das mesmas, existe uma carência de tais sistemas considerando-se o ambiente de programação Java. Finalmente, a última seção deste capítulo apresentou J2ME, com especial ênfase à configuração conhecida como CLDC, tendo sido destacado o fato de não ser possível utilizar a implementação tradicional de Java RMI neste ambiente, pois ele não suporta a reflexão computacional presente nas outras edições da linguagem Java. Com o intuito de validar Arcademis, este arcabouço foi utilizado no desenvolvimento de um serviço de invocação remota de métodos para CLDC/J2ME.

Capítulo 3

Descrição Geral de Arcademis

Este capítulo descreve as principais partes de plataformas de *middleware* orientadas por objetos e como tais partes estão implementadas em Arcademis. São descritos os principais padrões de projeto utilizados na implementação de cada uma destas partes e como elas podem ser alteradas de modo a permitir que o *middleware* se adapte a diferentes requisitos. No final deste capítulo é apresentada a classe ORB, um componente central de Arcademis, que determina as características semânticas de toda instância deste arcabouço.

3.1 Arquitetura de Arcademis

Arcademis permite o desenvolvimento de plataformas de *middleware* baseadas em objetos distribuídos, os quais se comunicam via invocação remota de métodos. Conforme discutido no Capítulo 2, um serviço de chamada remota de métodos permite que os desenvolvedores de aplicações utilizem objetos que não se localizam no mesmo espaço de endereçamento via uma sintaxe muito semelhante àquela utilizada para a programação não distribuída. Todo o processamento necessário para tratar a comunicação entre objetos é feito pelo sistema de *middleware* que suporta as aplicações distribuídas.

As plataformas de *middleware* obtidas a partir de Arcademis podem ser configuradas de diferentes formas, porém todas elas possuem um conjunto comum de classes e interfaces que definem a estrutura geral de um sistema baseado em invocação remota de métodos. Dois destes componentes comuns a qualquer instância de Arcademis são *stubs* e *skeletons*. O papel de um *stub* é transmitir os parâmetros de uma invocação remota para o objeto remoto ao qual a chamada se destina. Além disto, o *stub* se encarrega de receber o resultado da invocação, quando ele existe. Vê-se, portanto, que o *stub* deve possuir a mesma interface que o objeto remoto que ele representa, o que caracteriza um padrão de projeto conhecido como *proxy* [GHJV94]. O *skeleton*, por sua vez, é responsável por receber invocações remotas e repassá-las ao seu destino, ou seja, para o objeto que irá efetivamente executar tais chamadas. Também é papel do *skeleton* enviar para o cliente de uma invocação remota o resultado da mesma, caso este exista. Uma vez que o

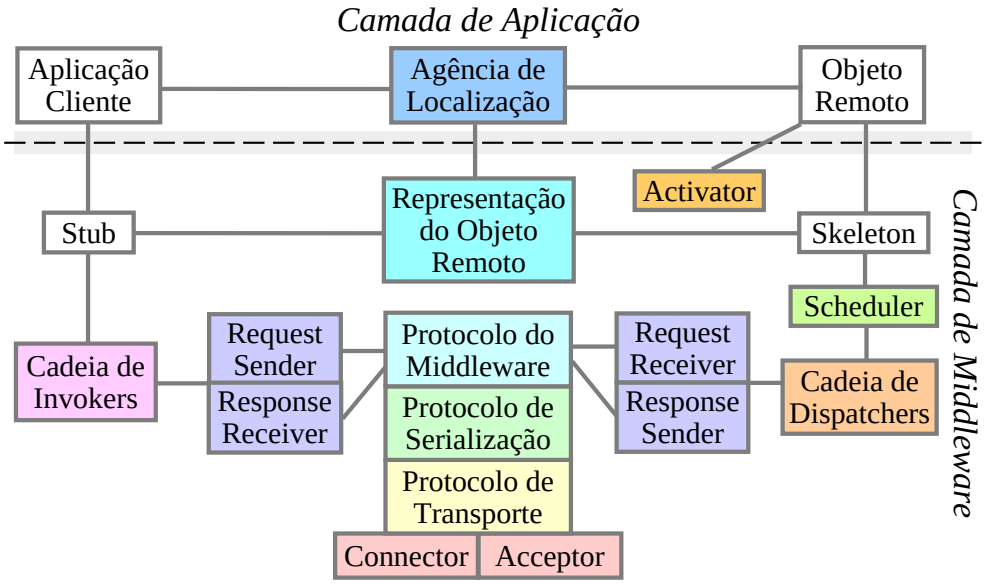

Figura 3.1: Principais componentes de Arcademis.

skeleton decodifica uma requisição remota e a repassa para a operação correspondente do objeto destino, ele pode ser caracterizado por um padrão de projetos denominado *adapter* [GHJV94].

Além de *stubs* e *skeletons*, uma plataforma de *middleware* derivada de Arcademis utiliza diversos outros componentes que visam garantir a comunicação entre objetos remotos e aplicações clientes. Tais componentes são descritos na Figura 3.1. Nesta figura, alguns módulos representam componentes individuais, enquanto outros representam conjuntos de componentes que interagem para fornecer algum tipo de funcionalidade ao desenvolvedor de aplicações. Os traços entre módulos representam colaborações entre componentes, porém nem todas elas aparecem na figura, a fim de que não seja comprometida a sua legibilidade.

Conforme pode ser inferido da Figura 3.1, invocações remotas não são emitidas diretamente por *stubs*, mas por meio de componentes denominados *invokers*. Do lado da aplicação servidora, o componente análogo ao *invoker*, que é responsável por receber invocações da rede e enviá-las para o objeto remoto, é denominado *dispatcher*. Chamadas remotas podem ser ordenadas de acordo com sua prioridade, o que é feito por um componente denominado `Scheduler`. A camada de rede propriamente dita é representada por componentes que compõem o protocolo do *middleware*, o protocolo de serialização e o protocolo de transporte. Para que o cliente possa estabelecer conexões, ele utiliza um componente de nome `Connector`, ao passo que do lado da aplicação servidora, conexões são recebidas por um componente conhecido como `Acceptor`. A interface entre *invokers* e a camada de rede é feita por dois componentes: `RequestSender` e `ResponseReceiver`. Já a interface entre *dispatchers* e a rede é feita pelos componentes `RequestReceiver` e `ResponseSender`. Finalmente, `Activator` é o componente responsável por inicializar objetos distribuídos e torná-los aptos a receberem invocações remotas.

Arcademis divide uma plataforma de *middleware* orientada por objetos em treze partes. Cada uma destas partes é responsável por executar uma função específica do *middleware*, como por exemplo, a inicialização de objetos remotos ou a serialização de tipos estruturados quando estes são passados como parâmetros de invocações. Contudo, nem todas as plataformas de *middleware* apresentam os treze aspectos definidos por Arcademis. Por exemplo, alguns sistemas podem utilizar um escalonador para entregar as chamadas remotas às entidades responsáveis por processá-las. Por outro lado, caso todas as chamadas possuam a mesma prioridade, então o escalonador não é necessário, e as requisições podem ser passadas para a implementação do objeto remoto assim que forem recebidas. Os treze constituintes básicos especificados por Arcademis são descritos a seguir:

- **Transporte de dados:** componentes deste grupo definem os mecanismos utilizados para transmitir dados entre aplicações clientes e servidoras.
- **Estabelecimento de conexões:** componentes deste grupo determinam como conexões são estabelecidas entre clientes e servidores. Para este fim, Arcademis utiliza o padrão *acceptor-connector* [Sch96].
- **Tratamento de eventos:** componentes deste grupo estipulam como o sistema de *middleware* reconhece e trata eventos, como, por exemplo, o estabelecimento de conexões ou a chegada de dados via um canal de comunicação.
- **Estratégia de Serialização:** componentes deste grupo definem como objetos podem ser convertidos em seqüências de *bytes* e como eles podem ser reconstruídos a partir de tais seqüências.
- **Protocolo do *Middleware*:** componentes deste grupo determinam o conjunto de mensagens que caracteriza as iterações entre objetos distribuídos.
- **Semântica de chamadas remotas:** este parâmetro estipula o nível de garantia que a plataforma de *middleware* fornece para uma aplicação cliente acerca da execução de chamadas remotas. Na Figura 3.1 os componentes que implementam este parâmetro são representados pelos elementos: *request sender*, *request receiver*, *response sender* e *response receiver*.
- **Representação de objetos remotos:** componentes deste grupo definem como objetos remotos são representados em um sistema distribuído.
- **Serviço de localização de nomes:** esta funcionalidade define como objetos remotos podem ser encontrados em um sistema distribuído. Na Figura 3.1, tal parâmetro é representado pelo componente denominado *agência de localização*.

- **Ativação do objeto remoto:** componentes responsáveis por este aspecto determinam como o objeto remoto é ativado, isto é, quais estruturas devem ser inicializadas e como isto deve ser feito para que tal objeto possa receber invocações remotas. Na Figura 3.1, a estratégia de ativação é representada pelo componente *activator*.

- **Estratégia de invocação:** este parâmetro define como uma chamada remota é invocada. Objetos adaptadores, denominados *invokers* podem ser acoplados entre o *stub* e os processadores de serviço.

- **Despacho de requisições:** este parâmetro determina como o servidor de chamadas está organizado e como ele aloca *threads* para a execução de chamadas remotas. Na Figura 3.1, esta estratégia é representada pelo componente denominado *dispatcher*.

- **Política de prioridades:** este parâmetro determina como o servidor distribui prioridades entre os métodos remotos. Em Arcademis, prioridades são distribuídas por um componente denominado *scheduler*.

- **Tipos de exceções:** este parâmetro define as situações anômalas que podem ocorrer em uma plataforma de *middleware*.

O restante deste capítulo descreve com maiores detalhes cada uma das partes de um sistema de *middleware* relacionadas nesta seção.

3.1.1 Transporte de Dados

O acesso às primitivas de baixo nível necessárias à transmissão de mensagens é encapsulado por dois componentes de Arcademis: a interface `Channel` e a classe abstrata `ConnectionServer`. `Channel` é responsável pelo envio e recebimento de dados, e sua implementação pode utilizar diferentes protocolos de comunicação, por exemplo, TCP, UDP ou HTTP. A classe `ConnectionServer`, por sua vez, implementa as funcionalidades necessárias para receber conexões. Assim como `Channel`, este componente pode ser implementado de diferentes maneiras, porém, quando usados em conjunto, objetos destes dois tipos precisam possuir implementações compatíveis. A fim de melhor ilustrar o papel destes componentes no estabelecimento de conexões, pode-se considerar que a interface `Channel` possui uma função análoga à classe `Socket` do pacote `java.net`, ao passo que a classe `ServerSocket` deste mesmo pacote corresponde à classe `ConnectionServer`. As codificações de `Channel` e `ConnectionServer` podem ser vistas na Figura 3.2.

Para representar a localização de entidades distribuídas, Arcademis fornece a interface `Epid` (cujo nome deriva do inglês *end point identifier*). Pontos de localização podem ser implementados de diferentes formas. Por exemplo, de acordo com o protocolo TCP/IP, tais componentes são definidos por um par, dado por um endereço IP e um número de porta. Exemplos de outras possíveis implementações são endereços de memória, em aplicações que se comunicam via

```
package arcademis.server;
import arcademis.*;
public abstract class ConnectionServer {
  protected Epid epid = null;

  public ConnectionServer(Epid epid) {
    this.epid = epid;
  }

  public abstract void accept()
  throws NetworkException;

  public abstract Channel getChannel();

  public abstract void
  setConnectionTimeout(int t);

  public abstract int
  getConnectionTimeout();
}
```

```
package arcademis;
public interface Channel {
  public void connect(Epid epid)
  throws NetworkException;

  public void send(byte[] a)
  throws NetworkException;

  public byte[] recv()
  throws NetworkException;

  public void close()
  throws NetworkException;

  public Epid getLocalEpid();

  public void setConnectionTimeout(int t);

  public int getConnectionTimeout();
}
```

Figura 3.2: A classe `ConnectionServer` e a interface `Channel`.

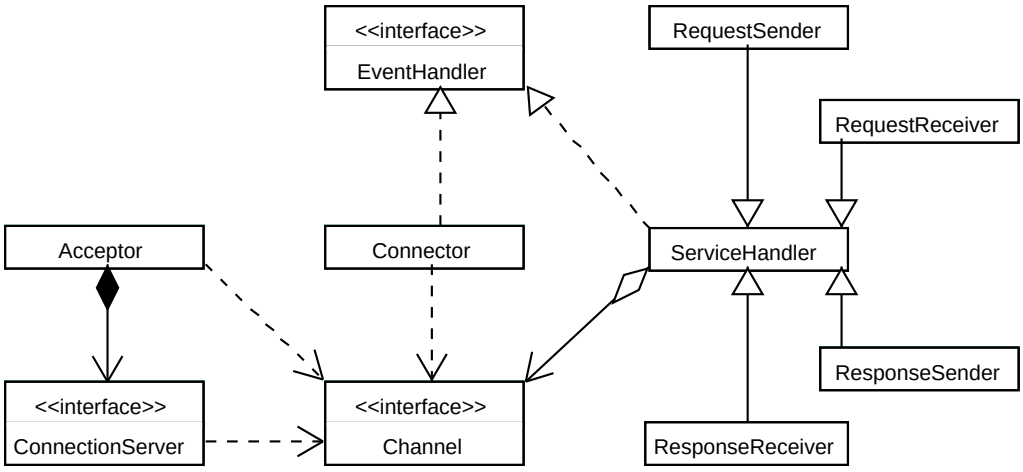

Figura 3.3: Principais componentes do padrão *acceptor-connector*.

memória compartilhada, ou nomes de arquivos, os quais podem ser dados por URLs acrescidas de um caminho em uma árvore de diretórios.

3.1.2 Estabelecimento de Conexões

A tarefa de estabelecer canais de comunicação entre servidores e clientes fica a cargo de um conjunto de objetos implementados segundo um padrão de projeto denominado *acceptor-connector* [Sch96]. A principal vantagem deste padrão é separar o estabelecimento de uma conexão de sua utilização, ou seja, os componentes responsáveis por criar conexões, segundo este padrão, não são as mesmas entidades responsáveis por transmitir dados por um canal, uma vez que o mesmo tenha sido criado. As entidades mais importantes que fazem parte deste padrão de projeto são mostradas na Figura 3.3.

```
public abstract class Acceptor
implements EventHandler {
  protected ConnectionServer cs = null;
  public void handleEvent(Event e) {}
  public Acceptor(Epid epid) {
    this.cs = OrbAccessor.
    getConnectionServer(epid);
  }
  public Acceptor(ConnectionServer cs) {
    this.cs = cs;
  }
  public abstract void
  accept(ServiceHandler h)
  throws NetworkException;
}
```

```
public class BlockingAcceptor
extends Acceptor {
  public void accept(ServiceHandler h)
  throws NetworkException {
    super.connectionServer.accept();
    Channel ch =
      super.connectionServer.getChannel();
    h.open(ch);
  }
  public BlockingAcceptor(Epid epid) {
    super(epid);
  }
  public BlockingAcceptor
  (ConnectionServer cs){
    super(cs);
  }
}
```

Figura 3.4: Classe `Acceptor` e uma possível implementação.

```
public abstract class Connector
implements EventHandler {
  private int timeout = 0;

  public abstract void connect
  (ServiceHandler h, Epid epid)
  throws NetworkException;

  public void handleEvent(Event e) {}

  public int
  getDefaultConnectionTimeout() {
    return timeout;
  }

  public void
  setDefaultConnectionTimeout(int t){
    timeout = t;
  }
}
```

```
public class SynchronousConnector
extends Connector {
  public void connect
  (ServiceHandler h, Epid epid)
  throws NetworkException {
    Channel ch = OrbAccessor.getChannel();
    ch.connect(epid);
    h.open(ch);
  }
}
```

Figura 3.5: Classe `Connector` e uma possível implementação.

De acordo com o padrão *acceptor-connector*, o estabelecimento de uma conexão fica a cargo de dois componentes. Do lado do cliente, existe uma entidade denominada *connector* que é responsável por, ativamente, iniciar o processo de criação de um canal de comunicação. No espaço de endereçamento do servidor existe uma entidade denominada *acceptor* cujo papel é receber a requisição de conexão, via um objeto do tipo `ConnectionServer`, e concluir a criação do canal. Tanto receptores quanto conectores, em Arcademis, são representados por classes abstratas, as quais podem ser vistas nas Figuras 3.4 e 3.5, ao lado de possíveis implementações.

Tendo sido concluído com sucesso o processo de criação de um canal, este é passado para o componente que efetivamente o utilizará para transmitir e receber informações. Tal componente, segundo o padrão *acceptor-connector*, é denominado *service handler*, ou processador de serviços e sua interface é mostrada na Figura 3.6. Duas diferentes implementações para processadores de serviços são mostradas na Figura 3.7. Nesta figura, o programa à esquerda envia uma mensagem e aguarda pela confirmação da mesma. O programa à direita representa a entidade

```
public abstract class ServiceHandler
       implements EventHandler {
  protected Channel channel = null;
  public abstract void open(Channel ch);
  public void handleEvent(Event e) {}
  public Channel getChannel() {
    return channel;
  }
}
```

Figura 3.6: Classe `ServiceHandler`.

```
public class Sender
extends ServiceHandler {
  public void open(Channel ch) {
    try {
      ch.send("Msg".getBytes());
      String s =
      new String(ch.recv());
      System.out.println(s);
    } catch (NetworkException e) {
      e.printStackTrace();
    }
  }
}
```

```
public class Receiver
extends ServiceHandler {
  public void open(Channel ch) {
    try {
      String s =
      new String(ch.recv());
      System.out.println(s);
      ch.send("Ack".getBytes());
    } catch (NetworkException e) {
      e.printStackTrace();
    }
  }
}
```

Figura 3.7: Duas implementações de processadores de serviço.

complementar: que, ao receber uma mensagem, envia a confirmação.

Conforme dito anteriormente, o principal benefício do padrão *acceptor-connector* é separar o processamento necessário ao estabelecimento de uma conexão do processamento necessário à transmissão de mensagens. Esta última atividade é implementada pelo método `open` da classe `ServiceHandler` ao passo que o estabelecimento de conexões fica a cargo dos métodos `accept`, da classe `Acceptor` e `connect`, da classe `Connector`. Assim, ao desenvolver um processador de serviços, o programador assume que este componente terá acesso a uma conexão já estabelecida, não importando se tal conexão foi criada, por exemplo, segundo um processo síncrono ou assíncrono. Da mesma forma, o programador, ao definir implementações para `Acceptor` e `Connector` não leva em conta, por exemplo, que tipos de mensagens são transmitidas pelos canais criados por tais componentes.

Um exemplo de utilização do padrão *acceptor-connector* pode ser visto na Figura 3.8. O programa mostrado à esquerda simplesmente se conecta a um servidor, e, por meio de um processador de serviço da classe `Sender`, mostrada como exemplo na Figura 3.7, envia uma mensagem e aguarda uma resposta para a mesma. O programa mostrado à esquerda na Figura 3.8 perfaz o papel de servidor: recebe mensagens e as responde com uma seqüência de *bytes*, via um processador de serviços da classe `Receiver` (Figura 3.7).

Os componentes `Acceptor` e `Connector` podem ser configurados de diversas formas, por exemplo, para que conexões sejam estabelecidas assincronamente, isto é, sem que a aplicação cliente permaneça bloqueada durante o processo de estabelecimento de conexão. Além disto,

```
public class Client {
  public static void main(String args[])
  throws Exception {
    String host = args[0];
    int port = Integer.parseInt(args[1]);
    Epid epid = new TcpEpid(host, port);
    ServiceHandler rs = new Sender();
    Connector c=new SynchronousConnector();
    c.connect(rs, epid);
  }
}
```

```
public class Server {
  public static void main(String a[])
  throws Exception {
    String host = a[0];
    int port = Integer.parseInt(a[1]);
    Epid e = TcpEpid(host, port);
    Acceptor ba=new BlockingAcceptor(e);
    while(true) {
    ServiceHandler r = Receiver();
    ba.accept(r);
  }
}
```

Figura 3.8: Aplicação cliente/servidora baseada no padrão *acceptor-connector*.

```
public abstract class Event {
  private Identifier uid;
  protected EventHandler eventHandler;
  public abstract Object getHandle();
  public abstract void initiate();
  public EventHandler getEventHandler() {
    return eventHandler;
  }
  public  Event(EventHandler event) {
    this.eventHandler = event;
    this.uid = OrbAccessor.getIdentifier();
  }
  public boolean equals(Object obj) {
    if(!(obj instanceof Event))
      return false;
    else
      return this.uid.equals
      (((Event)obj).uid);
  }
}
```

```
public interface EventHandler {
  public void handleEvent(Event e);
}
```

Figura 3.9: Componentes para o tratamento de eventos.

é possível, modificando-se a implementação de `Connector`, reutilizar canais de comunicação. Alterações na implementação de `Acceptor`, por sua vez, permitem definir, por exemplo, se novas *threads* são criadas para receber cada conexão ou não.

3.1.3 Tratamento de Eventos

Arcademis utiliza um modelo de tratamento de eventos baseado no padrão *reactor* [Sch94]. Eventos, modelados pela classe `Event`, podem ser utilizados para denotar situações tais como o estabelecimento de uma conexão ou a presença de dados em uma área de memória compartilhada. Toda entidade do tipo `Event` possui uma referência para um objeto do tipo `EventHandler`, o qual é responsável pelo tratamento de um ou mais eventos. Dessa forma, quando um objeto do tipo `Event` percebe a ocorrência de um evento, ele notifica o tratador de eventos para o qual possui referência. A classe `Event` e a interface `EventHandler` são mostradas na Figura 3.9.

A classe `Event` é abstrata porque ela não define o quê de fato constitui um evento. Possíveis eventos podem ser a passagem de um intervalo de tempo, o estabelecimento de uma conexão ou uma determinada resposta obtida em uma consulta a um banco de dados. Essa classe também não define como eventos são percebidos. Possíveis alternativas são a inspeção constante, técnica

esta conhecida como *pooling*, ou a notificação via interrupções do sistema operacional. Um terceiro parâmetro de configuração é a política de *threads* utilizada: eventos podem ser tratados na mesma *thread* onde ele foi notificado, ou para cada ocorrência de evento uma nova *thread* pode ser criada para seu tratamento. É possível ainda que todos os eventos sejam agrupados em uma estrutura centralizada, e uma única *thread* exclusiva para a notificação de eventos se encarregue de verificar a ocorrência dos mesmos.

Caso o desenvolvedor de *middleware* escolha usar um sistema centralizado para a notificação de eventos, ele pode lançar mão de um componente de Arcademis denominado *notifier*, onde eventos podem ser registrados. Essa estrutura pode ser configurada de diferentes formas. Por exemplo, o *notifier* pode utilizar uma única *thread* de controle para verificar a ocorrência de todos os eventos nele registrados, ou pode criar uma nova *thread* para cada evento. Esta estrutura adiciona certa complexidade ao sistema, porém não precisa ser utilizada em todas as instâncias do arcabouço. O *notifier* permite que diferentes tratadores de eventos sejam notificados de forma homogênea.

Eventos podem ser utilizados para o tratamento de conexões assíncronas. Com esse objetivo, a classe `Connector` implementa a interface `EventHandler`. Dessa forma, é possível que um conector relegue a uma *thread* separada o processo de estabelecimento de conexão. Quando esta *thread* terminar sua tarefa, ela notifica o conector, via o método `handleEvent()`, que a conexão já se encontra estabelecida. Tendo recebido a notificação, o conector ativa o processador de serviço que irá utilizar o canal de comunicação recém criado.

3.1.4 Estratégia de Serialização

Um serviço de invocação remota de métodos que não permite a passagem de objetos como parâmetros de chamadas ou como resultados das mesmas é muito limitado, pois apenas tipos primitivos tais como números inteiros ou valores booleanos poderiam ser utilizados em invocações remotas. A fim de permitir que objetos sejam utilizados em métodos remotos, é necessário que o *middleware* disponibilize ao desenvolvedor de aplicações uma maneira de serializá-los. Serializar um objeto [AGH00] significa transformá-lo em uma seqüência de *bytes*, a qual pode ser escrita em um arquivo ou transmitida por um canal de comunicação.

A linguagem Java utiliza, para serializar objetos, uma técnica conhecida como reflexividade [AGH00], a qual permite, durante a execução de um programa, que sejam conhecidos alguns aspectos internos à estrutura do mesmo, como, por exemplo, o tipo dinâmico de um objeto, ou a lista dos métodos que fazem parte de uma classe. Uma vez que Arcademis foi proposto como uma alternativa adequada a diversos tipos de ambientes de execução, alguns dos quais não suportam o mecanismo de reflexividade, o desenvolvedor de sistemas de *middleware* não deve contar com a possibilidade de utilizar tal técnica.

A fim de permitir que objetos sejam utilizados em invocações remotas, Arcademis transfere para a implementação de cada objeto a tarefa de serializá-lo. Com este intuito foi definida a

interface `Marshalable`, que caracteriza objetos serializáveis. Tal interface possui dois métodos, os quais são denominados `marshal` e `unmarshal`. O primeiro permite que um objeto especifique a seqüência de *bytes* que o descreve. O segundo deles, por sua vez, preenche os atributos internos do objeto a partir de uma seqüência de *bytes*.

Toda instância de *middleware* obtida a partir de Arcademis deve também definir um protocolo que descreve como tipos primitivos e objetos devem ser serializados e desserializados. Para tanto, este arcabouço declara uma interface denominada `Stream`, a qual contém a definição dos principais métodos utilizados no processo de serialização, como aqueles necessários para transformar números inteiros ou caracteres em seqüências de *bytes*. Um elemento do tipo `Stream` encapsula uma seqüência de *bytes* que descreve objetos e tipos primitivos, e pode ser implementado de diversas formas diferentes, por exemplo, via arranjos circulares de *bytes*, ou por meio de arquivos permanentes.

3.1.5 Protocolo do *Middleware*

O protocolo de comunicação adotado pela plataforma de *middleware* define o conjunto de mensagens que as aplicações distribuídas podem usar para se comunicarem. Por exemplo, para que uma invocação remota possa acontecer, é necessário que as aplicações clientes e servidoras troquem uma certa quantidade de mensagens, sendo que cada mensagem carrega informações próprias e origina respostas específicas por parte da entidade receptora. Em geral, cada sistema de *middleware* utiliza um protocolo específico. Java RMI, por exemplo, usa um protocolo denominado JRMP. CORBA, por sua vez, utiliza um protocolo denominado GIOP/IIOP e RME, a instância de Arcademis apresentada no Capítulo 4 utiliza um protocolo denominado RMEP. É possível permitir a interação entre diferentes plataformas de *middleware*, desde que elas façam uso de um protocolo comum. Por exemplo, aplicações desenvolvidas em Java RMI podem se comunicar com sistemas distribuídos baseados em CORBA devido a existência de uma implementação daquela plataforma sobre o protocolo IIOP [Mic03].

Em instâncias de Arcademis, o protocolo de comunicação é representado por uma classe denominada `Protocol` e por um conjunto de classes que implementam a interface `Message`. A classe `Protocol`, que pode ser vista na Figura 3.10, encapsula um elemento do tipo `Channel`, e como este, possui os métodos `send` e `receive`. Contudo, ao contrário da interface `Channel`, a classe `Protocol` lida com objetos do tipo `Message`, e não com cadeias de *bytes*.

Mensagens são objetos serializáveis, isto é, implementam a interface `Marshalable`, e utilizam os métodos `marshal` e `unmarshal` para determinar a seqüência de *bytes* que constitui a sua representação na camada de transporte. Um exemplo de implementação de mensagem pode ser visto na Figura 3.11. A classe mostrada neste exemplo designa uma mensagem que encapsula uma invocação remota de métodos em RME. Mensagens deste tipo contêm uma representação da chamada remota, o que inclui seu identificador, argumentos e endereço para retorno.

A criação e transmissão de mensagens é controlada pelos quatro processadores de serviço

```
package arcademis;
public class Protocol {

  private Channel ch = null;

  public void send(Message msg)
  throws ProtocolException,
  NetworkException {
    if (ch == null)
      throw new ProtocolException();
    try {
      Stream b = OrbAccessor.getStream();
      b.write(msg);
      ch.send(b.getBytes());
    } catch (MarshalException e) {
      throw new ProtocolException();
    }
  }

  public void setChannel(Channel ch) {
    this.ch = ch;
  }

  public Message recv()
  throws ProtocolException,
  NetworkException {
    if (ch == null)
      throw new ProtocolException();
    Message msg = null;
    try {
      Stream b = OrbAccessor.getStream();
      b.fill(ch.recv());
      msg = (Message)b.readObject();
    } catch (MarshalException e) {
      e.printStackTrace();
      throw new ProtocolException();
    }
    return msg;
  }

  public Channel getChannel() {
    return this.ch;
  }
}
```

Figura 3.10: A classe `Protocol`.

```
public class CallMsg implements Message {
  private RmeCall r = null;

  public void setRemoteCall(RmeCall r) {
    this.r = r;
  }

  public void marshal(Stream b)
  throws MarshalException {
    // send header: RMEP
    b.write(0x524D4550);
    // send protocol version:
    b.write((byte)0x01);
    // send message type: 0x50
    b.write(CALL_MESSAGE);
    // send epid:
    b.write(r.getReturnAddress());
    // send call identifier:
    b.write(r.getCallIdentifier());
    // send target remote object id:
    b.write(r.getTargetObjectId());
    // send operation code
    b.write(r.getOperationCode());
    // append the argument value:
    b.append(r.getArguments());
  }

  public RmeCall getRemoteCall() {
    return this.r;
  }

  public void unmarshal(Stream b)
  throws MarshalException {
    int header = b.readInt();
    if(header != 0x524D4550)
      throw new MarshalException();
    byte version = b.readByte();
    if(version != (byte)0x01)
      throw new MarshalException();
    byte type = b.readByte();
    if(type != (byte)CALL_MESSAGE)
      throw new MarshalException();
    // reading the end point identifier
    Epid h=(HostPortEpid)b.readObject();
    // reading the identifier of the call
    Id cuid=(Id)b.readObject();
    // reading the the target object id
    Id rID = (Id)b.readObject();
    // reading the operation code
    int opCode = b.readInt();
    // the remaining stream now
    // just holds the call's arguments
    this.r = new RmeCall(rID, b, opCode);
    this.r.setReturnAddress(h);
  }
}
```

Figura 3.11: Exemplo de implementação de mensagem.

```
public void open(Channel ch) {
  super.protocol.setChannel(ch);
  Message msg = super.protocol.recv();
  if(msg instanceof CallMsg) {
    RmeCall remoteCall = (RmeCall)((CallMsg)msg).getRemoteCall();
    Stream returnValue = dispatcher.dispatch(remoteCall);
    this.sendReturnValue(remoteCall.getCallIdentifier(), returnValue, super.protocol);
  } else if (msg instanceof PingMsg) {
    AckMsg ack = (AckMsg)OrbAccessor.getMessage(RmeConstants.PING_ACKNOWLEDGE);
    super.protocol.send(ack);
  } else {
    String unknowClassName = msg.getClass().getName();
    throw new ProtocolException("Wrong message type");
  }
}
```

Figura 3.12: Método `open` de `RmeRequestReceiver`.

apresentados na Seção 3.1.6. Na Figura 3.12 é mostrado o método `open` de um *request receiver* utilizado na implementação de RME. Tal método verifica o tipo de cada mensagem recebida a fim de tratá-la de forma adequada. A estrutura utilizada por Arcademis para representar o protocolo de comunicação facilita a alteração do mesmo, seja via inserção, remoção ou modificação da semântica de mensagens. Para adicionar novas mensagens ao protocolo, basta criar mais classes que implementam `Message` e inserir as rotinas apropriadas para a transmissão das mesmas nos quatro processadores de serviço. Para remover mensagens, remove-se os comandos que as tratam do corpo dos processadores de serviço e destrói-se as classes que as implementam. Por fim, para alterar a semântica de uma chamada, basta modificar o código da classe que a implementa para adicionar ou remover informações. Uma vez que mensagens possuem tipo bem definido, é possível utilizar métodos sobrecarregados em superclasses de `Protocol` para tratar de forma diferente mensagens específicas.

3.1.6 Semântica de Chamadas Remotas

Em um ambiente distribuído, a execução de uma chamada remota está sujeita a diferentes contingências. Por exemplo, atrasos na rede podem fazer com que uma requisição demore muito a ser processada, ou que se perca antes de atingir o servidor. A resposta originada da execução de um método remoto também pode não alcançar o processo cliente que solicitou sua execução. Dado este cenário sujeito a inúmeras contingências, diferentes níveis de garantias podem ser fornecidos às aplicações distribuídas pelo sistema de *middleware*. Tais níveis de garantia determinam diferentes semânticas para chamadas remotas. Algumas das semânticas mais comuns são: *best effort*, *at least once*, *at most once* e *exactly once* [CDK96].

best effort Este é o modelo de tratamento de erros mais simples que pode ser implementado, sendo também conhecido como *maybe call semantics* [CDK96]. Dada uma chamada remota, o *middleware* garante que ela é executada no máximo uma vez, ou que não é executada. Não é necessário salvar qualquer tipo de informação sobre o estado da aplicação,

nem do lado cliente, tampouco no lado servidor da mesma.

at least once Esta semântica garante à aplicação cliente que toda chamada remota disparada é processada pelo servidor pelo menos uma vez, podendo, eventualmente, ser processada mais de uma vez. A fim de garantir esta semântica, em geral, quando o servidor recebe uma invocação remota, ele a responde para a aplicação cliente. Caso o cliente não receba resposta em um determinado período de tempo, ele pode repetir a chamada. Em aplicações em que métodos remotos não causam efeitos colaterais, esta é a semântica normalmente utilizada, pois reúne razoáveis níveis de garantia com relativa facilidade de implementação.

at most once A semântica conhecida como *at most once* garante às aplicações clientes que uma chamada remota é processada no máximo uma vez. Com este propósito, servidores mantém algum tipo de informação acerca das últimas chamadas atendidas e descartam requisições repetidas quando estas ocorrem. Dependendo da implementação fornecida, o servidor pode notificar ou não um cliente sobre o recebimento de chamadas repetidas. A semântica *best-effort* também garante ao desenvolvedor de aplicações que chamadas remotas são processadas no máximo uma vez, porém, ao contrário da semântica *at-most-once*, nova tentativa de invocação não é feita caso seja verificada uma falha de comunicação.

exactly once Este tipo de semântica garante que uma chamada remota é processada uma e somente uma vez. Em um ambiente distribuído, a implementação dessa semântica é extremamente custosa, e normalmente uma abordagem menos conservadora é adotada: caso a aplicação cliente venha a receber uma mensagem de confirmação, referente à execução de uma invocação remota, então ela pode garantir que a mensagem foi processada exatamente uma vez [HGM01].

A fim de permitir que suas instâncias utilizem diferentes semânticas, Arcademis utiliza um padrão de projeto denominado *request-response*. Este padrão é definido por quatro componentes interpostos entre o *stub* e o *skeleton*. Tais componentes são denominados *request-sender*, *request-receiver*, *response-sender* e *response-receiver* e se relacionam com *stubs* e *skeletons* de acordo com o esquema apresentado na Figura 3.13.

Dos quatro processadores de serviço citados anteriormente, dois são utilizados por aplicações clientes e os outros dois por aplicações servidoras. Do lado cliente fazem parte os componentes *request-sender* e *response-receiver*, sendo o primeiro empregado pelo *stub* para transmitir uma chamada remota e o segundo usado pelo *stub* para receber o resultado da invocação. Do lado servidor de uma aplicação distribuída, a entidade conhecida como *request-receiver* é utilizada para receber invocações remotas e repassá-las para o *skeleton*. A função de *response-sender* é enviar o resultado obtido via o processamento de uma chamada remota para a aplicação cliente que aguarda por ele.

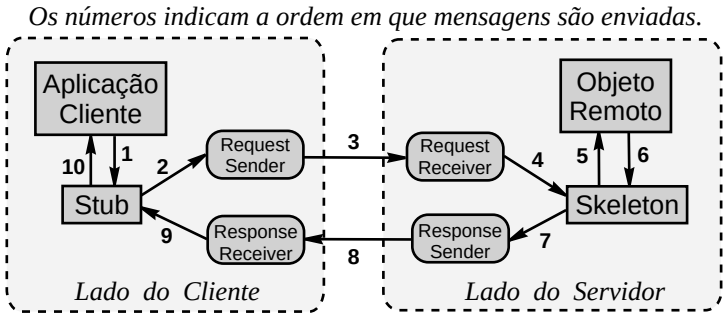

Figura 3.13: Processadores de serviço utilizados na comunicação entre *Stub* e *Skeleton*.

3.1.7 Ativação do Objeto Remoto

A estratégia de ativação determina quais recursos são alocados para o objeto remoto e como este se tornará apto a receber invocações remotas de métodos. Objetos distribuídos somente podem receber chamadas remotas após serem explicitamente ativados. Com este fim, a classe `RemoteObject` implementa a interface `Active`, que pode ser vista na Figura 3.14. Tal interface define dois métodos: `activate` e `deactivate`. O método `activate` é responsável por tornar o objeto remoto ativo, ou, em outras palavras, capaz de receber requisições remotas. A implementação do método `deactivate`, por seu turno, deve conter as rotinas necessárias à liberação, para o sistema operacional, de todos os recursos utilizados pelo objeto remoto. Embora a semântica deste método seja definida pelo desenvolvedor de *middleware*, existem algumas tarefas que normalmente são realizadas pelo método `activate`:

- definição de um ou mais pontos de localização para o objeto remoto, ou seja, o endereço no qual aquele objeto receberá invocações remotas;
- criação e inicialização de um receptor de conexões, conforme discutido na Seção 3.1.2;
- inicialização de um `Dispatcher`, que, conforme discutido na Seção 3.1.9, é responsável por repassar requisições para a real implementação do objeto remoto na forma de invocações de métodos.

A implementação dos métodos `activate` e `deactivate` pode ser alterada para modificar a política de *threads* adotada pelo servidor. Por exemplo, `activate` pode determinar que o servidor seja executado em uma *thread* separada ou na mesma linha de execução que o invocou. Aspectos estruturais da arquitetura do servidor também podem ser configurados neste método. Por exemplo, `activate` pode associar ao objeto remoto um `Dispatcher` que repassa mensagens diretamente para o *skeleton*, ou pode utilizar uma versão daquele componente que insere mensagens em uma fila de prioridades.

```
package arcademis.server;

public interface Active {

  public void activate()
  throws ActivationException;

  public void deactivate()
  throws ActivationException;

}
```

Figura 3.14: A interface `Active`.

A ativação do objeto remoto pode ser implementada diretamente no corpo do método `activate` deste objeto. Por outro lado, Arcademis fornece para o desenvolvedor de *middleware* um componente denominado `Activator`, que se presta justamente a este fim. Ativadores são criados por uma fábrica associada ao ORB e visam facilitar possíveis reconfigurações sobre a estratégia de inicialização de objetos remotos.

3.1.8 Estratégia de Invocação

A estratégia de invocação adotada pela plataforma de *middleware* determina como requisições são despachadas da aplicação cliente para o servidor responsável por recebê-las. Em Arcademis tal estratégia é implementada por um componente denominado `Invoker` e por seus decoradores. São tarefas do invocador criar um *request sender* para transmitir requisições até a implementação do objeto remoto e receber o valor de retorno originado pelo processamento das mesmas. A maneira como um invocador é implementado permite configurar diversos aspectos da plataforma de *middleware*, tais como a semântica da chamada remota (o invocador pode escolher componentes entre diversas implementações de processadores de serviço), o reaproveitamento de conexões entre invocações sucessivas e quais funcionalidades extras são utilizadas por cada invocação (caches, geradores de *log*, *buffers*, etc).

Conforme mostrado na Figura 3.15, todo *stub* possui referência para pelo menos um objeto do tipo `Invoker`. Arcademis, contudo, permite que um *stub* agrupe diversos invocadores diferentes, a fim de que diferentes níveis de qualidade de serviço possam ser fornecidos para aplicações distribuídas. Por exemplo, a fim de aumentar a eficiência de invocações remotas, é possível definir invocadores que realizam chamadas do tipo *one way*, que não exigem confirmação por parte do servidor. Neste caso, não é necessário que a aplicação permaneça bloqueada durante o processamento da invocação remota. Embora chamadas do tipo *one way* possam ser tratadas pelo sistema de *middleware* de forma bastante eficiente, chamadas do tipo *two way* são mais comuns em sistemas tradicionais, como Java RMI. Neste caso, a aplicação cliente necessita aguardar por uma confirmação do servidor de invocações acerca da execução de cada chamada remota.

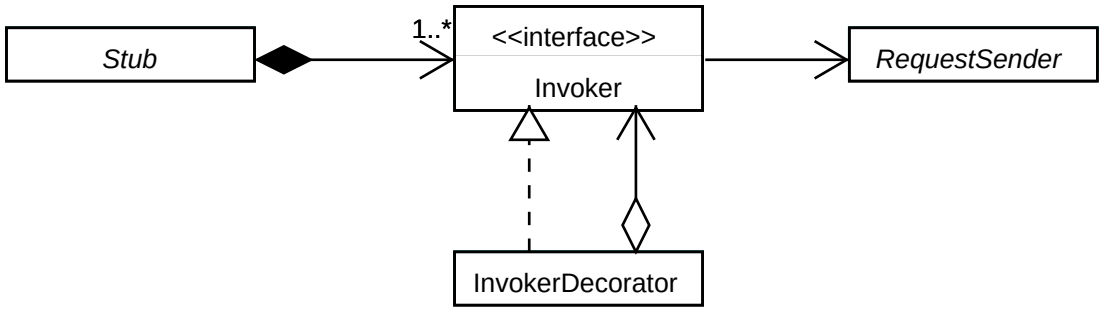

Figura 3.15: Relação entre os componentes *stub* e `Invoker`.

Arcademis fornece um decorador de invocadores a fim de permitir que funcionalidades extra sejam agregadas a estes componentes. Dessa forma, pode-se compor cadeias de invocadores, onde cada invocador é responsável pela implementação de uma nova capacidade, como por exemplo:

- o uso de *caches* a fim de reaproveitar resultados de invocações remotas realizadas previamente, conforme tratado na Seção A.6;
- a monitoração do desempenho das aplicações clientes;
- o balanceamento de carga. Neste caso, a plataforma de *middleware* deve permitir que invocadores contenham referência para mais de um servidor de invocações remotas;
- a simulação de anomalias na rede. Pode-se utilizar, por exemplo, um invocador que corrompa, atrase ou destrua aleatoriamente o conteúdo de algumas mensagens, com o objetivo de simular as condições de redes sem fio, por exemplo.
- o enfileiramento de mensagens, a fim de melhor aproveitar a banda de transmissão de dados disponível, conforme discutido na Seção A.7.

3.1.9 Despacho de Requisições

A estratégia de despacho define como uma invocação remota é passada para a implementação do objeto remoto responsável por processá-la, uma vez que ela tenha sido recebida no lado servidor de uma aplicação distribuída. Tal estratégia, em Arcademis, é implementada por um componente denominado `Dispatcher`, o qual pode ser customizado a fim de determinar diferentes estruturas para o servidor de requisições. Normalmente o `Dispatcher` repassa as requisições recebidas diretamente para o *skeleton*, que por sua vez as transmite para a implementação do objeto remoto. Este tipo de organização é adotado, por exemplo, na implementação de Java RMI e pode ser observado na Figura 3.16. Outros arranjos, contudo, são possíveis. Com o objetivo de permitir que a plataforma de *middleware* distribua diferentes prioridades entre as requisições

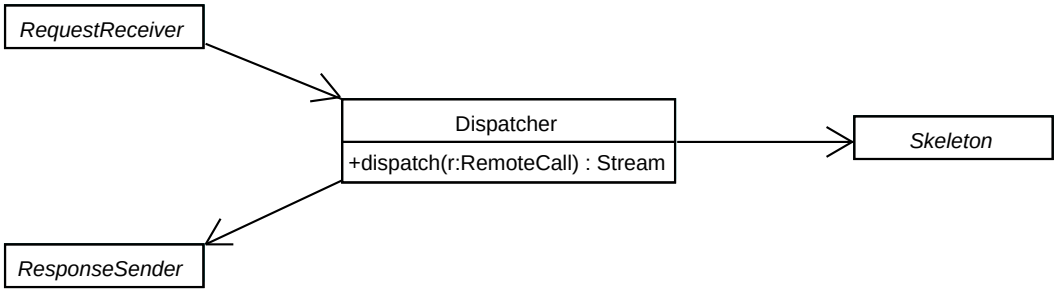

Figura 3.16: `Dispatcher` diretamente conectado ao *skeleton*.

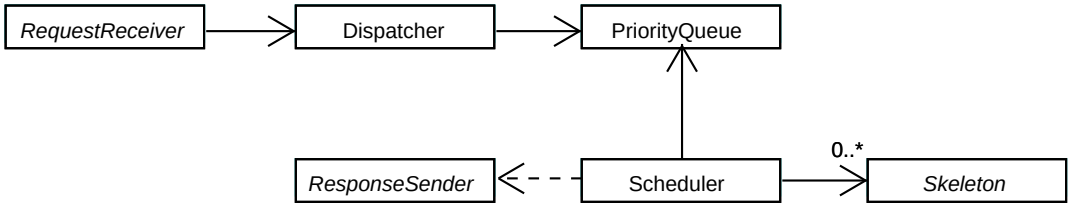

Figura 3.17: `Dispatcher` conectado à fila de prioridades.

recebidas, o `Dispatcher` pode inserir as requisições recebidas em uma fila de prioridades, em vez de repassá-las diretamente para o *skeleton*. Tal organização é mostrada na Figura 3.17.

Caso seja necessário agregar funcionalidades extra a um *Dispatcher*, Arcademis fornece a classe *DispatcherDecorator*, a qual implementa o padrão de projeto conhecido como *decorador* [GHJV94]. Exemplos de tais funcionalidades são a geração de arquivos de *log* para fins de monitoração do servidor e a verificação da autenticidade das mensagens recebidas, isto é, se elas de fato foram transmitidas por um *stub* válido. A interface *Dispatcher* e sua classe decoradora podem ser vistas na Figura 3.18.

A implementação do `Dispatcher` determina também a política de *threads* adotada pelo

```
package arcademis.server;

import arcademis.*;

public interface Dispatcher {
  public abstract Stream
  dispatch(RemoteCall remoteCall);
}
```

```
package arcademis.server;

import arcademis.RemoteCall;
import arcademis.Stream;

public class DispatcherDecorator
implements Dispatcher {
  protected Dispatcher d;

  public DispatcherDecorator(Dispatcher d){
    this.d = d;
  }

  public Stream dispatch(RemoteCall c){
    return this.d.dispatch(c);
  }
}
```

Figura 3.18: A interface *Dispatcher* e a classe *DispatcherDecorator*.

servidor. Diversas políticas são possíveis, como atribuir uma *thread* separada para a execução de cada chamada remota, ou utilizar um banco de *threads* para o processamento das mesmas. Outros componentes também podem ser utilizados a fim de customizar a política de *threads*. Por exemplo, a implementação de *request receiver* presente em RME cria uma nova *thread* para cada conexão estabelecida com uma aplicação cliente. Caso seja utilizado o arranjo mostrado na Figura 3.17, o escalonador, implementado pelo componente denominado `Scheduler`, também pode ser customizado para utilizar diferentes políticas de *threads*.

3.1.10 Política de Prioridades

Conforme discutido na Seção 3.1.9, um servidor de invocações remotas pode ser estruturado de diferentes formas. Em um dos possíveis arranjos, requisições não são passadas diretamente para o *skeleton*, mas inseridas em uma fila de prioridades. Uma entidade denominada escalonador se encarrega de remover requisições da fila e repassá-las para o *skeleton* apropriado. A fim de permitir que servidores sejam projetados segundo este tipo de arquitetura, Arcademis fornece ao desenvolvedor de *middleware* a interface de dois componentes: `Buffer` e `Scheduler`.

O componente denominado `Buffer` representa filas de prioridade, ou seja, estruturas onde requisições de chamadas remotas podem ser inseridas e que permitem a configuração da ordem em que tais itens são retirados. A prioridade de uma chamada é um dos argumentos da classe `RemoteCall`. Existem diversos modos de distribuir prioridades entre chamadas remotas. Uma distribuição de prioridades usual é processar as requisições na ordem em que são recebidas. Tal política é conhecida como FIFO (*First In, First Out*). Outras políticas de distribuição de prioridades existem, por exemplo: associar uma prioridade diferente a cada método remoto, atribuir uma prioridade particular a cada cliente ou monitorar o tempo de execução dos métodos para determinar a prioridade dos mesmos. O restante desta seção descreve três exemplos de política de distribuição de prioridades que podem ser utilizadas pelo desenvolvedor de *middleware*.

A fim de atribuir um valor de prioridade para cada método de um objeto remoto, em Arcademis, basta modificar a implementação do *stub*. O *stub* fornece uma implementação para cada método remoto, e portanto, pode ser adaptado para atribuir a eles valores particulares de prioridade. Por exemplo, métodos cuja execução demanda, em média, menor tempo podem receber maior prioridade, ao passo que, a métodos que realizam tarefas maiores, cabem prioridades menores. Esta estratégia é conhecida como *Shortest Job First* [SG98] e é ótima, no sentido de garantir o menor tempo de espera na fila. Em servidores que recebem muitas requisições diferentes, sendo o tempo aproximado de execução de cada método previamente conhecido, é a melhor política de prioridades que pode ser adotada.

Outra possibilidade é atribuir prioridades diferentes para clientes diferentes. Isto pode ser feito de diversas formas. Uma possível alternativa é fazer com que invocações de métodos passem a ser "assinadas". Uma chamada assinada contém informações particulares de cada cliente, que podem ser usadas como um identificador único. A prioridade de cada chamada pode ser definida

pelo *dispatcher*, ou por um decorador para este tipo de componente. Uma questão pertinente, neste caso, é qual o critério adotado para distribuir prioridades entre clientes. Uma solução é monitorar a freqüência com que cada cliente invoca métodos remotos e assinalar prioridades a cada chamada de acordo com esta freqüência.

A terceira política de prioridades, e a mais complexa dentre os três exemplos citados, consiste em atribuir prioridades a endereços remotos. Nesse caso, um servidor deve poder receber requisições em mais de um endereço, sendo cada endereço distinto alocado a um diferente objeto do tipo *Acceptor*, mostrado na Seção 3.1.2. Os *request receivers* criados por cada receptor de conexões devem possuir um valor pré-definido de prioridade para atribuir às chamadas remotas.

Duas questões pertinentes à terceira política de prioridades são, em primeiro lugar, como fazer com que clientes diferentes tenham acesso ao mesmo servidor via pontos de localização distintos e, em segundo lugar, qual o critério para informar a localização do servidor para cada cliente. A fim de mostrar uma possível solução para cada pergunta, considere um serviço de nomes que informe respostas diferentes de acordo com a origem do cliente. Clientes que se localizam na mesma instituição que o servidor recebem maior prioridade. Para verificar a localização de cada cliente, o servidor pode utilizar o endereço IP do mesmo. Todo servidor, neste caso, possui duas localizações armazenadas na agência de localização, uma de alta prioridade, e outra de baixa.

Este arranjo envolve a alteração da agência de localização e dos receptores de conexões. Em primeiro lugar, a agência de localização deve ser modificada para informar respostas diferentes de acordo com a origem do cliente. Em segundo lugar, receptores de conexões, representados por componentes do tipo `Acceptor` e `RequestReceiver`, precisam ser alterados para atribuir prioridades às chamadas remotas.

3.1.11 Representação de Objetos Remotos

Todas as instâncias de plataformas de *middleware* derivadas de Arcademis apresentam alguns componentes comuns. Dois destes componentes são as *referências remotas* e os *objetos remotos*. A relação entre tais entidades é mostrada na Figura 3.19. Em um *middleware* orientado por objetos, é necessário que aplicações clientes tenham como referenciar objetos que estão localizados em outros espaços de endereçamento. A fim de permitir este tipo de referência, o arcabouço fornece uma classe denominada `RemoteReference`, que representa uma referência remota, ou seja, uma referência para um objeto que não se encontra disponível no espaço de endereçamento local. Na Figura 3.19, vê-se que tanto objetos remotos quanto *stubs* possuem um atributo do tipo `RemoteReference`. Este é o único elo entre implementações de objetos remotos e os *stubs* que os representam no espaço de endereçamento de processos clientes.

Referências remotas guardam duas informações básicas: a localização do objeto remoto e um conjunto de valores que o identificam de maneira única em uma rede de computadores. A fim de representar a localização de um objeto remoto, ou seja, o ponto sobre o qual conexões com o dito objeto podem ser estabelecidas, Arcademis fornece a interface `Epid`. Diversas informações

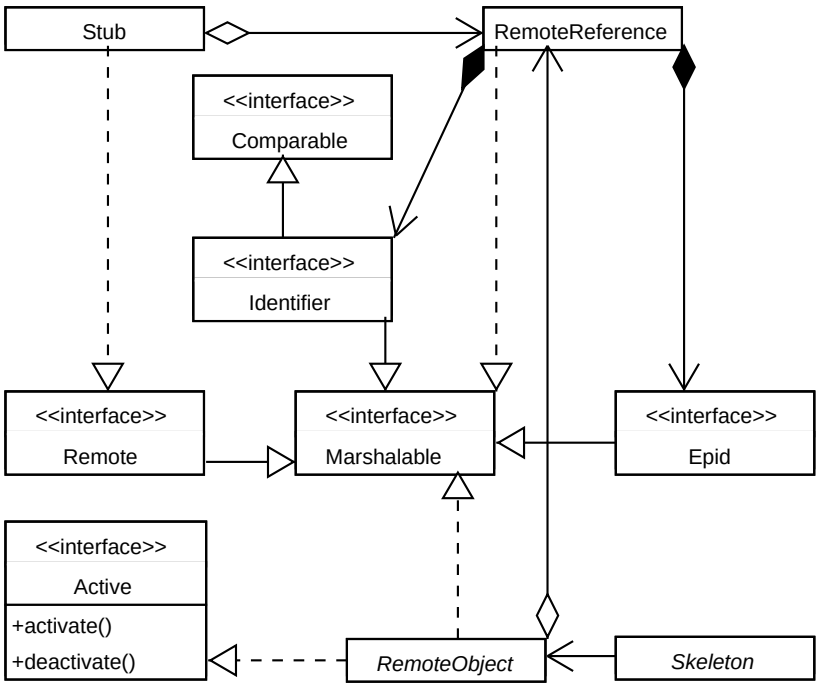

Figura 3.19: Objetos remotos e referências remotas.

diferentes podem ser utilizadas para caracterizar um ponto de localização, por exemplo, o par formado pelo endereço IP e o número de porta, o endereço de um arquivo em uma hierarquia de diretórios, ou o identificador de uma área de memória compartilhada. Cabe ao usuário do arcabouço fornecer uma implementação adequada à interface `Epid`.

Assim como pontos de localização, identificadores de objetos distribuídos também são definidos por Arcademis como uma interface, a qual é denominada `Identifier`. Em instâncias de Arcademis, identificadores devem ser implementados com o propósito de discriminar de modo não ambíguo cada objeto remoto presente em um sistema distribuído. Em Java RMI [Mic03], por exemplo, objetos são identificados por quatro valores. O primeiro deles define o momento em que o objeto foi criado. O segundo valor representa o código da máquina virtual em que o objeto foi instanciado, sendo único em um mesmo *host*. O terceiro valor é um número inteiro de 64 bits que diferencia objetos criados na mesma máquina virtual em instantes de tempo muito próximos. O último valor é o código *hash* do objeto remoto.

De acordo com a especificação de Arcademis, todo objeto remoto, isto é, todo objeto capaz de receber invocações remotas, deve implementar a classe abstrata `RemoteObject`, ou uma de suas classes herdeiras. Tal implementação é responsável por definir a semântica de métodos tais como `toString()`, `hashCode()` e `equals()`. A semântica de tais métodos pode variar dependendo da localização real do objeto. A semântica remota destes métodos é definida pela classe `RemoteReference`, portanto cabe ao desenvolvedor de *middleware* decidir se referências remotas e locais apresentam o mesmo comportamento em relação a tais operações.

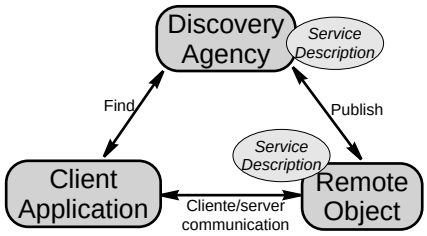

Figura 3.20: Arquitetura orientada por serviços.

3.1.12 Serviço de Localização de Nomes

Plataformas de *middleware* orientadas por objetos em geral baseiam-se em uma arquitetura orientada por serviços [BHTV03]. Esse tipo de modelo envolve três tipos diferentes de entidades: *fornecedores de serviços*, *clientes* e *agências de localização*. Fornecedores de serviços são entidades que detêm alguma funcionalidade que pode ser utilizada pelas partes clientes que compõem o sistema. Com o intuito de permitir que clientes tenham como encontrar os serviços disponíveis, fornecedores precisam publicar uma descrição das funcionalidades que eles disponibilizam, o que é feito via *agências de localização*. Tais agências fornecem, basicamente, duas operações, que no contexto desta seção, são chamadas `publish` e `find`. Uma representação tradicional deste tipo de arquitetura pode ser vista na Figura 3.20.

O modelo apresentado na Figura 3.20 pode ser implementado de diferentes formas. Por exemplo, em Java RMI o usuário tem acesso à agência de localização por meio de uma classe denominada `Naming` [Mic03]. Esta classe permite que objetos remotos sejam registrados em um diretório onde são identificados por nomes únicos. A cada um destes nomes está associado um *stub*. Em Java RMI as operações `publish` e `find` são chamadas, respectivamente, de `lookup` e `bind`. O método `lookup` recebe como parâmetro uma URL, que contém o endereço de um *host* e o nome do objeto procurado. Caso tal objeto esteja presente no *host* especificado, esta operação retorna para a aplicação cliente o *stub* associado àquele objeto. O método `bind`, por sua vez, recebe dois parâmetros: um *stub* e o nome ao qual tal componente deve ser associado junto à agência de localização.

É possível implementar agências de localização que forneçam para o usuário interfaces diferentes daquela oferecida por Java RMI. Por exemplo, em vez de objetos remotos serem representados por nomes únicos, eles podem ser identificados diretamente pelos serviços que oferecem, isto é, por suas interfaces. Neste caso, o método `publish` deve associar, junto à agência de localização, cada objeto remoto ao nome da interface implementada por ele. O método `find` deve receber como parâmetros o endereço de um *host* e a interface que descreve os serviços procurados. Para tratar tentativas de registro de objetos que implementam interfaces iguais no mesmo diretório de nomes, existem diferentes alternativas. Pode-se por exemplo, permitir que tais eventos ocorram e, diante de buscas, qualquer implementação adequada é retornada. Por

```
package arcademis.concrete;
import arcademis.*;

public interface NameBasedNaming {
  public Remote find(String name)
  throws NotBoundException,
         ArcademisException;
}
```

```
package arcademis.concrete.server;
import arcademis.*;
import arcademis.server.*;

public interface NameBasedNaming {
  public void publish
  (String name, Stub obj)
  throws AlreadyBoundException,
  ArcademisException;
}
```

Figura 3.21: Definição de agência de localização baseada em nomes.

```
package arcademis.concrete;
import arcademis.*;

public interface
InterfaceBasedNaming {
  public Remote find(Class service)
  throws NotBoundException,
         ArcademisException;
}
```

```
package arcademis.concrete.server;
import arcademis.*;
import arcademis.server.*;

public interface
InterfaceBasedNaming {
  public void
  publish(RemoteObject obj)
  throws AlreadyBoundException,
  ArcademisException;
}
```

Figura 3.22: Definição de agência de localização baseada em tipo.

outro lado, pode-se restringir o diretório de nomes de forma que somente uma implementação seja registrada por interface.

Devido ao fato de ser possível definir o serviço de localização de nomes de diferentes formas, Arcademis não predefine nenhuma interface para a agência de localização. Assim, tais entidades devem ser implementadas de forma independente pelo desenvolvedor de *middleware*. Por outro lado, em sua coleção de componentes concretos, Arcademis define duas interfaces diferentes para a agência de localização, as quais podem ser vistas nas Figura 3.21 e 3.22. A primeira figura mostra um serviço de localização semelhante ao que é utilizado por Java RMI, sendo os objetos remotos identificados por nomes únicos. A Figura 3.22 apresenta um serviço de localização onde objetos são pesquisados com base nas interfaces que eles implementam. Em ambos os casos, as operações `publish` e `find` foram declaradas em interfaces diferentes porque dessa forma uma aplicação não precisa implementar um dos métodos se não for precisar dele. Por exemplo, uma aplicação cliente necessita somente da interface que declara o método `find` e, assim, não precisa implementar a operação `publish`. Da mesma forma, um servidor de métodos remotos pode não precisar de utilizar o método `find`, e, neste caso, deverá prover somente uma implementação para a operação `publish`.

Também fazem parte da semântica de um serviço de nomes os diferentes tipos de mensagens de erro que podem ser disparadas durante a sua utilização. Erros podem ocorrer, por exemplo, quando determinada consulta não encontra qualquer resultado válido, ou quando ocorrem conflitos devido à colisões de nomes durante a operação de registro. Arcademis define três tipos de erros, os quais são representados pelas classes de exceção: `AlreadyBoundException`, `NotBoundException` e `InvalidAccessException`. A primeira exceção caracteriza a tentativa

de utilização de um nome que já foi usado em um registro anterior. O segundo tipo de erro caracteriza a não ocorrência de um nome solicitado em uma operação de busca no diretório de nomes. Finalmente, a terceira exceção qualifica o evento em que um cliente tenta localizar um nome, este se encontra registrado, porém o cliente não possui permissão para invocar métodos sobre o objeto remoto ao qual o nome se refere.

3.1.13 Tipos de Exceções

É possível catalogar diversos tipos diferentes de situações excepcionais que podem ocorrer em uma plataforma de *middleware*. Situações deste tipo ocorrem, por exemplo, quando um conector tenta efetuar uma conexão em um ponto no qual não existe nenhum receptor ativo, ou quando acontece uma tentativa de leitura de uma seqüência de *bytes* que não descreve corretamente um objeto com base no protocolo de serialização adotado. A seguir podem ser vistos alguns tipos de exceções previstas por Arcademis. Além destas exceções, Arcademis define outros tipos de erros, relacionados ao serviço de nomes, conforme apresentado na Seção 3.1.12.

ActivationException Essa exceção deve ser levantada quando o objeto remoto não pode ser corretamente ativado. Por exemplo, caso um objeto remoto tente criar um receptor de conexões em uma porta que já é utilizada por outra aplicação, durante a sua ativação, então uma exceção desse tipo deve ser disparada.

MalformedAddressException Esse tipo de exceção deve ser disparada quando o *middleware* se depara com a especificação de um endereço que não obedece à sintaxe adotada. Por exemplo, caso endereços sejam definidos via URIs (*Uniform Resource Identifiers*), então um endereço como `turmalina.dcc.ufmg.br!8080/obj` deveria levantar uma exceção deste tipo, pois o sinal `!` não pertence ao conjunto de símbolos usados na declaração de URIs.

MarshalException Esse tipo de exceção ocorre quando não é possível serializar um objeto que implementa `Marshalable` ou quando não é possível reaver um objeto desse tipo a partir de uma cadeia de *bytes*. Por exemplo, tal exceção deveria ser disparada quando acontecer uma tentativa de ler um inteiro de uma seqüência de menos que quatro *bytes*, uma vez que, na linguagem Java, inteiros possuem 32 *bits*.

NetworkException Essa exceção ocorre devido à falhas na camada de transporte, em geral detectadas na implementação da interface `Channel`. Por exemplo, ela deveria ser disparada quando acontecer uma tentativa de conexão a um ponto no qual não exista nenhum receptor ativo.

ProtocolException Esse tipo de erro caracteriza a ocorrência de mensagens inesperadas de acordo com o protocolo de comunicação adotado. Uma falha desse tipo acontece quando uma mensagem com o cabeçalho inválido é recebida.

```
package arcademis;

public interface ChannelFc {
  public Channel createChannel();
  public Channel createChannel(int t);
}
```

```
import arcademis.*;
public class ExChFc implements ChannelFc{
  public Channel createChannel() {
    return new DefaultChannel();
  }
  public Channel createChannel(int type){
    switch(t) {
      case TCP: return new TcpChannel();
      case UDP: return new UdpChannel();
      case SOAP: return new SoapChannel();
      default: return createChannel();
    }
  }
}
```

Figura 3.23: Fábrica de canais de comunicação.

ReconfigurationException Esse tipo de exceção deve ocorrer sempre que for detectada uma tentativa de reconfigurar o ORB tendo essa estrutura já sido configurada. Conforme é discutido na Seção 3.2, após a configuração do ORB, novas reconfigurações não são permitidas.

UnspecifiedException Essa exceção é retornada por um servidor para o cliente quando, durante o processamento de um método remoto, uma exceção que não implementa a interface `Marshalable` é detectada. Exceções disparadas por métodos remotos devem implementar a interface `Marshalable` para que seja possível informar ao cliente sobre a exata natureza do erro que foi detectado.

IncompatibleStubException Essa exceção deve ser disparada quando um objeto remoto recebe uma chamada proveniente de um *stub* que não foi criado por ele. Em Arcademis, a implementação de objetos remotos pode ser caracterizada como uma fábrica de *stubs*. Um *Stub* é gerado contendo o identificador do objeto remoto que o originou. Cada invocação remota de métodos deve incluir no corpo da mensagem que a constitui o identificador do *proxy* de onde ela provêm, de modo que, no espaço de endereçamento do servidor, os identificadores do *stub* e do servidor possam ser comparados.

3.2 A classe ORB

A fim de facilitar a tarefa de reconfiguração, a maior parte dos componentes de Arcademis não são criados diretamente, mas a partir de fábricas de objetos, de acordo com o padrão de projeto conhecido como *fábrica* [GHJV94]. Uma fábrica de objetos é um componente que retorna uma, dentre possíveis instâncias de um objeto, de acordo com as informações que lhe são fornecidas [Coo00]. A interface para a fábrica de canais pode ser vista na Figura 3.23. Arcademis define 17 fábricas, as quais estão relacionadas na Tabela 3.1. Caso necessário, o desenvolvedor de *middlewares* pode adicionar fábricas extras.

O acesso a cada fábrica de objetos que compõem Arcademis se dá via uma estrutura deno-

Fábrica	Componente	Fábrica	Componente
AcceptorFc	Acceptor	NotifierFc	Notifier
ChannelFc	Channel	RemoteRefFc	RemoteReference
ConnectionServerFc	ConnectionServer	SchedulerFc	Scheduler
ConnectorFc	Connector	ServiceHandlerFc	ServiceHandler
EpidFc	Epid	DispatcherFc	Dispatch
IdentifierFc	Identifier	StreamFc	Stream
BufferFc	Buffer	InvokerFc	Invoker
ProtocolFc	Protocol	MessageFc	Message
ActivatorFc	Activator		

Tabela 3.1: Conjunto de fábricas fornecidas pelo arcabouço.

Figura 3.24: A classe ORB e algumas fábricas de objetos.

minada *broker*, a qual é representada pela classe ORB, cuja implementação obedece um padrão de projetos denominado *singleton* [GHJV94]. Segundo a definição deste padrão [Coo00], dada uma classe, esta pode ter somente uma instância, ou, doutro modo, não pode ser instanciada, sendo utilizada via métodos estáticos, tal qual se dá, por exemplo, com a classe Math do pacote java.lang. A classe ORB segue o mesmo princípio adotado na implementação da classe Math: sua declaração contém o modificador final, o que impede sua especialização via herança; todos os seus métodos são estáticos e seu método construtor foi declarado como privado, o que proíbe sua instanciação. Uma representação parcial da classe ORB é mostrada na Figura 3.24.

Todas as aplicações distribuídas baseadas em Arcademis devem fornecer uma configuração para o *broker* antes de iniciarem sua execução. A configuração do *broker* consiste na definição de quais fábricas farão parte dele, o que, de certa forma, determina que instância de *middleware* será gerada. Uma vez definida a configuração do *broker*, ela não mais poderá ser modificada durante a execução da aplicação. Tal fato previne a realização de alguns tipos de reconfigurações dinâmicas, pois as fábricas inicialmente adotadas não podem mais ser modificadas. Existem casos, entretanto, em que mudanças durante a execução de aplicações são possíveis. Por exemplo, nada impede que determinado algoritmo de criptografia ou compactação passe a ser utilizado em um canal, desde que as duas entidades comunicantes que o utilizam sejam capazes de negociar

```
package arcademis;

public interface Configurator {
  public void configure()
  throws ReconfigurationException;
}
```

```
import arcademis.*;

public final class ExConfigurator
  implements Configurator {
  public void configure() throws
  ReconfigurationException {
    // Define the channel factory
    ChannelFc cnFc = new RmeChFc();
    ORB.setChannelFactory(cnFc);
    // Define the connection
    // server factory
    ConnectionServerFc csFc=new RmeCSFc();
    ORB.setConnectionServerFactory(csFc);
    // Define the end point id factory
    IdentifierFc idFc = new RmeIdFc();
    ORB.setIdentifierFactory(idFc);
    // close the ORB for
    //further reconfigurations
    ORB.closeForReconfiguration();
  }
}
```

Figura 3.25: Interface para configuração do *broker*.

```
import arcademis.*;

public class Client {
    public static void main(String args[]) throws ReconfigurationException {
        ExConfigurator conf = new ExConfigurator();
        conf.configure();
        // start distributed application
    }
}
```

Figura 3.26: Trecho de código mostrando inicialização do ORB.

a técnica a ser adotada.

Para a configuração do *broker*, Arcademis define a interface `Configurator`, que é mostrada na Figura 3.25, junto com uma possível implementação. A classe ORB possui dois estados bem definidos: *aberto* e *fechado* para configuração. O estado inicial é sempre aberto; uma vez ativado o estado fechado, esta situação não pode mais ser modificada. Caso uma tentativa de reconfiguração aconteça estando o estado fechado ativo, uma exceção do tipo `ReconfigurationException` é disparada. Para que o estado fechado passe a estar ativo, o método `closeForReconfiguration` deve ser invocado sobre a classe `ORB`. Essa tarefa é responsabilidade do desenvolvedor de *middlewares* e normalmente é executada no corpo do método `configure`, conforme mostrado na Figura 3.25. A Figura 3.26 apresenta um trecho de código que mostra a configuração do *broker* antes do início da execução de uma aplicação.

A utilização de fábricas para a criação de objetos torna a derivação de novas plataformas de *middleware* a partir de sistemas antigos muito simples. Basta alterar a fábrica, ou as fábricas, responsáveis pela criação dos componentes cuja semântica se deseja alterar. Por exemplo, caso o desenvolvedor de *middlewares* deseje modificar o tipo de canal de comunicação utilizado em

uma certa plataforma, basta-lhe alterar a fábrica responsável pela criação de canais para que esta passe a retornar o novo canal. Nenhuma outra alteração é necessária nesse caso.

Uma desvantagem da criação de fábricas é que esse mecanismo adiciona ao sistema uma sintaxe longa que pode contribuir para a criação de programas pouco legíveis. Com o intuito de facilitar a tarefa do desenvolvedor de aplicações, Arcademis utiliza o padrão *Façade* [GHJV94] para encapsular tarefas repetitivas como a criação de componentes. Com esse fim, a classe `arcademis.OrbAccessor` fornece uma série de métodos estáticos que podem ser utilizados para a criação de componentes, protegendo, assim, os desenvolvedores da sintaxe extra adicionada pelo uso de fábricas. Exemplos de uso da fachada `OrbAcessor` podem ser vistos nas Figuras 3.4 e 3.5.

3.3 Conclusão

Este capítulo expôs uma visão geral acerca da arquitetura de Arcademis e como este arcabouço divide plataformas de *middleware* orientadas por objetos em partes mais básicas, tais como protocolos de comunicação e serialização, representação de objetos remotos e serviço de nomes. A divisão de sistemas de *middleware* em seus diversos constituintes é uma das principais contribuições desta dissertação, pois esta separação facilita a implementação e posterior reconfiguração de tais plataformas. Cada um dos trezes aspectos básicos em que a especificação de *Arcademis* divide sistemas de *middleware* foi apresentado em detalhes ao longo deste capítulo. Por fim, este capítulo apresentou a classe ORB, que agrupa em uma só entidade todas as fábricas usadas em Arcademis para a criação de componentes. A utilização de fábricas visa facilitar o processo de reconfiguração de plataformas de *middleware* obtidas de Arcademis: basta modificar fábricas específicas do sistema para alterar alguns de seus aspectos. Exemplos concretos de reconfigurações são apresentados no Capítulo 4 e no apêndice.

Capítulo 4

Estudo de caso: RME

Este capítulo apresenta RME, uma instância de Arcademis que fornece para a configuração CLDC de *Java 2 Micro Edition* (J2ME) um serviço de invocação remota de métodos. Além de ilustrar como Arcademis pode ser utilizado para o desenvolvimento de sistemas de *middleware*, a existência de RME representa, por si só, uma contribuição, uma vez que na literatura da área não existem descrições de outros serviços semelhantes desenvolvidos para CLDC.

4.1 Motivação

A demanda por aplicações voltadas para computadores móveis como PDA's e telefones celulares é cada vez maior. Recentes avanços na tecnologia empregada em tais dispositivos contribuíram para diminuir o custo dos mesmos e torná-los acessíveis a um grupo maior de pessoas. A título de exemplo, estima-se que existam mais de um bilhão de assinantes de serviços de comunicação sem fio em todo o mundo [RTV01]. Esses avanços também contribuíram para aumentar a capacidade computacional de dispositivos móveis, de modo que, atualmente são comuns aparelhos celulares capazes de utilizar informações disponíveis na Internet ou que podem comportar-se como clientes em aplicações de comércio eletrônico.

Ao contrário do mercado de computadores pessoais, para os quais existem poucos fabricantes, o universo dos dispositivos móveis, como telefones celulares, é extremamente segmentado. Existem diversos fabricantes e também vários padrões de comunicação, por exemplo, GSM, TDMA e CDMA. A grande variedade de processadores e protocolos existentes no mundo da computação sem fio compromete a portabilidade de programas entre dispositivos móveis e constitui um obstáculo aos desenvolvedores de aplicações, pois, sem um padrão universalmente adotado, dificilmente um aplicativo projetado para determinado tipo de processador pode ser utilizado em um aparelho diferente sem necessitar de modificações em seu código executável. A fim de amenizar os problemas causados pela grande variedade de fabricantes e protocolos existentes e fornecer aos projetistas de aplicações um modelo de desenvolvimento comum, foi desenvolvida a plataforma J2ME [RTV01], apresentada na Seção 2.6 do presente trabalho.

Dentre as características da plataforma J2ME, que a tornam adequada a uma ampla gama de dispositivos móveis, destacam-se seu tamanho reduzido, a portabilidade e a facilidade de escrita e manutenção de código que proporciona. J2ME [RTV01] é um ambiente de execução Java que requer menos de um décimo dos recursos necessários a J2SE, o sistema Java tradicionalmente utilizado no desenvolvimento de aplicações para computadores de mesa [AGH00]. O tamanho reduzido da plataforma é uma característica importante porque, mesmo com os recentes avanços na tecnologia de computação móvel, dispositivos sem fio são ainda limitados computacionalmente, sendo que muitas vezes não dispõem de mais que algumas dezenas de *kilobytes* de memória. A importância da portabilidade, por sua vez, deve-se à grande variedade de dispositivos móveis existentes, pois esta propriedade permite a reutilização de código entre diferentes processadores e facilita a comunicação entre sistemas heterogêneos. Por fim, Java, sendo uma linguagem de alto nível, orientada por objetos, possui melhores mecanismos de abstração que outras linguagens e ferramentas tradicionalmente utilizadas para o desenvolvimento de aplicações para sistemas embutidos, como C, por exemplo, o que facilita o projeto e manutenção de sistemas de *software*.

Conforme discutido na Seção 2.6, a plataforma J2ME apresenta diversas configurações, cada uma delas específica para um determinado grupo de processadores. A configuração destinada a aparelhos de telefonia celular, denominada CLDC [RTV01], é muito simples, e, embora disponibilize aos desenvolvedores de aplicações uma ampla variedade de funcionalidades, não contém muitos dos serviços tradicionalmente encontrados em outras versões da linguagem Java, como o mecanismo de Invocação Remota de Métodos. Tendo sido projetada para dispositivos severamente limitados em termos de recursos computacionais, a configuração CLDC não comporta os mecanismos de reflexividade que Java RMI utiliza para serializar objetos. Por outro lado, Java RMI não pode ser reconfigurado de modo a não depender de tais mecanismos, pois, como descrito na Seção 2.2.3, embora a plataforma RMI forneça ao desenvolvedor algumas opções de reconfiguração, a técnica adotada para a serialização de objetos não é uma delas.

Assim, com o intuito de fornecer ao usuário de CLDC um serviço de invocação remota de métodos facilmente reconfigurável, Arcademis foi utilizado para instanciar um *middleware* que não pressupõe a existência de qualquer mecanismo de reflexividade ou serialização fornecido pelo ambiente de execução. Esse sistema foi denominado RME (*RMI for J2ME*) [PVBB03].

4.2 Apresentação de RME

RME é um sistema de *middleware* orientado por objetos, que permite a invocação remota de métodos segundo uma sintaxe semelhante à utilizada na programação não-distribuída. Existem duas versões de RME implementadas. A primeira delas, `Server_RME`, destina-se à plataforma J2SE da linguagem Java, e permite que aplicações distribuídas usufruam das características e serviços providos por aquela edição da linguagem. A segunda versão, denominada `Client_RME`, foi desenvolvida para ser utilizada na configuração CLDC da plataforma J2ME e contém somente

as funcionalidades necessárias a aplicações clientes, isto é, `Client_RME` fornece às aplicações distribuídas a capacidade de invocar métodos sobre objetos remotos, mas não permite que as mesmas se comportem como aqueles objetos, processando invocações recebidas remotamente.

4.2.1 Arquitetura de Serviços

Assim como Java RMI, RME apresenta uma *arquitetura orientada por serviços* [BHTV03], a qual foi descrita na Seção 3.1.12. Em RME, fornecedores de serviços são representados pela implementação dos objetos remotos. A agência de localização é representada por um serviço de resolução de nomes, no qual objetos remotos podem registrar-se e que aplicações clientes podem consultar em busca de determinados nomes. Por fim, toda aplicação capaz de utilizar o diretório de nomes para obter informações sobre um objeto remoto e, a seguir, solicitar a execução de métodos remotos sobre esse objeto é um cliente.

Os serviços são métodos que podem ser invocados remotamente. Tais serviços são descritos em interfaces especiais, que estendem a interface `Remote` e, em sua declaração devem indicar a possibilidade de dispararem exceções do tipo `ArcademisException`. As características semânticas da interface `Remote` são as mesmas apresentadas pela interface de mesmo nome do pacote Java RMI. Em Java RMI, métodos remotos também indicam a possibilidade de serem disparadas exceções, as quais são do tipo `RemoteException`.

Fornecedores de serviços, em termos concretos, são objetos cuja classe implementa uma ou mais interfaces do tipo `Remote` e que estendem uma classe denominada `rme.server.RmeRemoteObject`. Como a linguagem Java permite que classes implementem múltiplas interfaces, um mesmo fornecedor de serviços pode prover às aplicações clientes diversos conjuntos diferentes de funcionalidades. Fornecedores de serviços são identificados por nomes, isto é, cadeias de caracteres. Em uma instância da agência de localização não podem haver nomes repetidos, e, em um dado *host* existe somente uma destas agências, logo, os nomes que designam objetos remotos são únicos em cada nodo de um sistema distribuído. Nodos, neste caso, são representados pelo par formado por um endereço IP e um número de porta.

A plataforma RME permite que aplicações desenvolvidas sobre a plataforma J2ME realizem invocações remotas de métodos, porém tais aplicações não podem fornecer serviços, possuindo apenas as funcionalidades necessárias a programas clientes. Fornecedores de serviços, em RME, são executados sobre a plataforma J2SE porque geralmente programas servidores demandam maior quantidade de recursos, sendo este um requisito que dificilmente poderia ser atendido no restrito ambiente que caracteriza os computadores móveis. Por exemplo, um servidor de serviços pode ter de lidar com dezenas de requisições simultâneas, e o tratamento de tal demanda não pode ser realizado satisfatoriamente por um processador incapaz de executar algo mais que 100 instruções por milisegundo.

4.3 A Arquitetura de RME

Durante o desenvolvimento de RME, Arcademis foi utilizado, em sua maior parte, como um arcabouço do tipo "caixa-branca", pois a maior parte dos componentes daquele *middleware* são implementações de classes abstratas e realizações de interfaces fornecidas por este arcabouço. Alguns componentes concretos de Arcademis, contudo, puderam ser aproveitados diretamente, ou seja, sem a necessidade de modificações. Dentre tais componentes citam-se os identificadores de objetos remotos e os pontos de localização utilizados como endereços para os mesmos. Tais componentes são mostrados na Seção 4.3.9. A lista a seguir resume como cada um dos treze aspectos enumerados na Seção 3.1 foram implementados em RME:

- **Transporte de dados:** RME utiliza o protocolo TCP/IP para a transmissão de mensagens.
- **Estabelecimento de conexões:** conexões são estabelecidas sincronamente, sempre de uma aplicação cliente para uma aplicação servidora.
- **Tratamento de eventos:** RME não implementa mecanismos de tratamento e notificação de eventos.
- **Protocolo do *Middleware*:** RME utiliza um protocolo denominado RMEP, o qual define quatro tipos de mensagens diferentes: *call*, *ping*, *ack* e *return*.
- **Estratégia de Serialização:** a classe `RmeStream` encapsula uma cadeia de *bytes* que contém informações serializadas.
- **Semântica de chamadas remotas:** RME pode ser configurado para utilizar as semânticas *best effort* ou *at most once*.
- **Representação de objetos remotos:** objetos são representados remotamente pela classe `RemoteReference` de Arcademis, e localmente pela classe `RmeRemoteObject`.
- **Serviço de localização de nomes:** RME utiliza a mesma interface do serviço de localização de nomes fornecida por Java RMI.
- **Ativação do objeto remoto:** em RME, objetos remotos são inicializados pela classe `RmeActivator`, que cria uma *thread* independente para o recebimento de chamadas remotas.
- **Estratégia de invocação:** chamadas remotas são invocadas sincronamente por uma implementação de `Invoker` que procura reaproveitar o canal de comunicação entre sucessivas chamadas remotas.
- **Despacho de requisições:** o componente `RmeDispatcher` repassa invocações remotas diretamente para o *skeleton*.

- **Política de prioridades:** RME não utiliza escalonadores, e invocações remotas são passadas para o *skeleton* assim que recebidas.
- **Tipos de exceções:** RME não define novos tipos de exceções e, para a notificação de situações excepcionais, utiliza as classes definidas em Arcademis.

4.3.1 Canais de Comunicação e Estabelecimento de Conexões

Os canais de comunicação de RME utilizam o protocolo TCP/IP para o envio e recebimento de mensagens. A interface `Channel` é implementada pela classe `TcpSocketChannel` e a interface `ConnectionServer` é implementada pela classe `TcpSocketServer`. Essa relação é mostrada na Figura 4.1.

Conforme discutido na Seção 3.1.2, o estabelecimento de conexões em Arcademis, e por conseguinte em RME, se dá segundo um padrão de projetos conhecido como *acceptor-connector*. O conector, em RME, procura estabelecer conexões sincronamente, isto é, uma vez iniciada a tentativa de estabelecimento de conexão, uma condição de erro é verificada sempre que a imediata efetivação da mesma não for possível. O receptor, por sua vez, permanece bloqueado a espera de conexões e as trata na medida em que são recebidas. O diagrama de classes que descreve as entidades responsáveis pelo estabelecimento de conexões é mostrado na Figura 4.1. As classes `BlockingAcceptor` e `SynchronousConnector` implementam, respectivamente, as funcionalidades de receptores e conectores.

Todos os quatro processadores de serviço previstos em Arcademis foram implementados em RME. Tais processadores, que foram descritos na Seção 3.1.6, são utilizados para enviar as mensagens que caracterizam uma invocação remota. Por razões de eficiência, cada invocação de método causa o envio de somente duas mensagens: uma requisição do cliente para o servidor e uma resposta em sentido contrário. As classes `RmeRequestSender` e `RmeRequestReceiver` são mostradas na Figura 4.1. Os outros dois processadores de serviço: `RmeResponseReceiver` e `RmeResponseSender`, são criados em fases posteriores do processo de invocação remota. O primeiro deles é instanciado assim que o cliente termina de enviar a chamada. O segundo processador, por seu turno, é criado logo que a chamada tenha sido processada pelo servidor.

A fim de tornar o processo de transmissão de dados mais eficientes, o canal de transmissão utiliza um *buffer* para fazer com que apenas mensagens completas sejam enviadas. Todas as mensagens trocadas pelo canal de comunicação são precedidas por um inteiro que descreve a quantidade de informação transmitida. A implementação das operações de leitura e escrita é mostrada na Figura 4.2. A fim de diminuir o número de acessos à camada de transporte, o tamanho do *buffer* é concatenado ao início da mensagem antes da mesma ser transmitida, e é lido como um arranjo de quatro elementos durante a operação de leitura. Testes realizados sobre a implementação do canal de comunicação mostraram que, caso o descritor do tamanho de cada mensagem fosse enviado antes da própria mensagem, em uma operação separada, então o tempo de transmissão de mensagens seria quase duas vezes o tempo de transmissão obtido

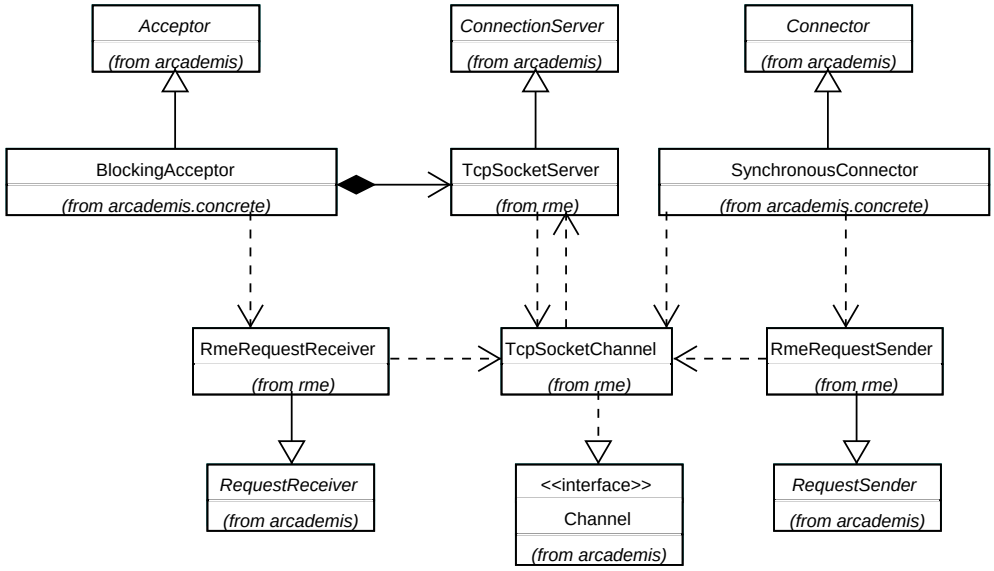

Figura 4.1: Classes utilizadas na implementação do padrão *acceptor-connector*.

```
public void send(byte[] msg) {
  byte[] b=new byte[msg.length+4];
  int bufSize = msg.length;
  b[0]=(byte)((bufSize>>24) & 255);
  b[1]=(byte)((bufSize>>16) & 255);
  b[2]=(byte)((bufSize>>8) & 255);
  b[3]=(byte)(bufSize & 255);
  for(int i=0;i<msg.length;i++)
    b[i+4] = msg[i];
  out.write(b);
}
```

```
public byte[] recv() {
  byte[] b = new byte[4];
  int status = in.read(b);
  if(status < 0) return null;
  int i4=(b[0]<<24)&0xFF000000;
  int i3=(b[1]<<16)&0xFF0000;
  int i2=(b[2]<<8)&0xFF00;
  int i1=b[3] & 0xFF;
  int bufSize = i1+i2+i3+i4;
  buf = new byte[bufSize];
  in.read(buf);
  return buf;
}
```

Figura 4.2: Implementação de `send` e `recv` em `TcpSocketChannel`.

com a implementação mostrada na Figura 4.2. Isto porque, uma vez que as mensagens não são muito grandes, elas podem ser transmitidas completamente em um único pacote TCP. Caso o tamanho das mensagens fosse enviado em uma operação separada, seria necessário repetir a mesma operação de transmissão de dados duas vezes para cada mensagem trocada.

4.3.2 Utilização de *Threads*

Aplicações clientes não criam novas *threads* durante a realização de uma chamada remota. Isso faz com que todas as invocações de métodos sejam síncronas, isto é, o programa cliente permanece bloqueado até que a efetivação de uma chamada remota seja possível. Servidores de serviço, por outro lado, criam uma nova *thread* para cada conexão estabelecida com um cliente. É importante notar que *threads*, em RME, não são criadas para tratar invocações remotas, mas conexões. Tal fato contribui para tornar o sistema mais eficiente, na medida em que diminui o número total de *threads* existentes e, por conseguinte, diminui o número de trocas de contexto e

a quantidade de recursos alocados pela máquina virtual Java para a manipulação de diferentes fluxos de execução.

Dado que a limitação do número de *threads* existentes melhora o desempenho do sistema, poder-se-ia perguntar porque não fazer com que o servidor de métodos remotos execute em uma única *thread*. A principal desvantagem desta abordagem é o fato dela não permitir que o processamento de chamadas remotas se dê em paralelo à realização de operações de transmissão de dados via a rede de comunicação. Ou seja, em uma aplicação com vários fluxos de execução, enquanto alguns realizam operações de entrada e saída, outros podem processar chamadas remotas. Por outro lado, se o número de *threads* em execução torna-se muito grande, então o tempo gasto pelo sistema operacional para efetuar trocas de contexto passa a ser relevante e impacta no desempenho da aplicação.

4.3.3 Reaproveitamento de Conexões

Na implementação de RME uma referência remota sempre denotará o mesmo objeto durante todo o seu ciclo de vida. A fim de garantir essa propriedade, objetos e referências remotas possuem um código identificador, o qual é enviado no cabeçalho de uma mensagem que caracteriza uma invocação remota. Esse identificador é inspecionado pelo *skeleton*, na máquina servidora, e caso se verifique uma inconsistência, uma exceção é retornada para o cliente. Dado um objeto remoto específico, a implementação de RME garante que seu identificador apenas é partilhado pelos *stubs* criados por ele.

Dado que durante o ciclo de vida de um *stub* o objeto remoto que ele representa é sempre o mesmo, é possível tentar reaproveitar conexões de rede a fim de melhorar o desempenho da comunicação entre clientes e servidores. Assim como em Java RMI, na implementação de RME esta abordagem foi adotada. Um *stub*, em RME, possui um atributo interno que representa um canal de comunicação. Enquanto o *stub* ainda não foi utilizado para enviar chamadas remotas, este canal é um atributo nulo. Logo que uma requisição é solicitada ao *stub*, na ausência de qualquer tipo de erro, um canal de comunicação é criado entre ele e o servidor remoto, conforme descrito na Seção 4.3.1. Esse canal, contudo, não é fechado imediatamente após a realização da invocação remota, pois ele pode ser reaproveitado caso outra requisição seja feita sobre o mesmo *stub*. A implementação de RME garante que, tendo sido realizada uma invocação remota, o canal utilizado por ela continuará existindo durante um determinado período de tempo, que é um valor constante da implementação de `Invoker` utilizada por RME. Findo esse período, uma invocação remota de método levará à criação de um novo canal de comunicação.

Conforme discutido na Seção 4.3.2, uma nova *thread* é criada no espaço de endereçamento do servidor para tratar toda conexão recebida. A fim de que objetos remotos e *stubs* utilizem políticas compatíveis, o canal de comunicação utilizado pelo servidor para tratar uma invocação não é destruído ao término de sua utilização. A implementação da classe `RmeRequestReceiver` mantém o canal recentemente utilizado aberto durante um determinado período de tempo, pois

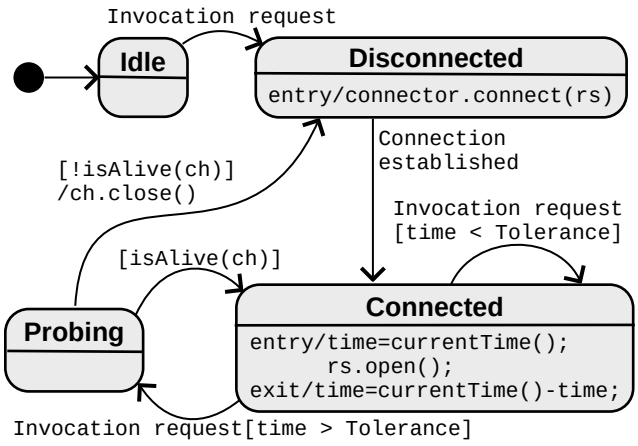

Figura 4.3: Diagrama de estados mostrando o reaproveitamento de conexões em RME.

a aplicação cliente pode utilizá-lo para efetuar novas requisições remotas. Assim como o intervalo de tempo utilizado na implementação do cliente, o tempo de espera adotado no lado servidor de uma aplicação é determinado em tempo de compilação. Fechar o canal após certo tempo é útil para liberar recursos e para evitar que a aplicação cliente permaneça indefinidamente bloqueada em face de erros de transmissão. Esta última situação pode ocorrer caso uma mensagem de retorno se perca enquanto transmitida do servidor para o cliente.

Retornando à implementação de `Invoker`, se uma nova invocação de métodos é solicitada, mas o período de tolerância já expirou, então uma tentativa de reutilizar o canal de comunicação existente é realizada. Essa tentativa consiste do envio de uma mensagem do tipo *ping* pelo canal. Uma mensagem desse tipo sempre origina uma resposta, caso a entidade para a qual ela foi destinada esteja ativa. Caso o *Ping* não origine uma resposta, então o o invocador passa a realizar os procedimentos necessários para a criação de um novo canal de comunicação. A máquina de estados que caracteriza a classe `TwoWayInvoker`, que implementa o invocador utilizado em RME, é mostrada na Figura 4.3.

A Figura 4.3 representa os estados em que o invocador pode se encontrar. Desde o momento de sua criação, até o momento em que a primeira invocação remota é realizada sobre o *stub*, ele se encontra no estado denominado `Idle`. Tendo recebido uma solicitação para efetuar uma chamada remota, o invocador passa para o estado `Disconnected`. Nesse estado, o canal de comunicação utilizado para a comunicação entre *stub* e objeto remoto ainda não foi criado. Assim que um canal é criado e a invocação acontece, o estado do invocador passa a ser denominado `Connected`. Enquanto neste estado, novas requisições podem utilizar o canal de comunicação sem a necessidade de testes, pois a implementação de RME garante que, na ausência de falhas de rede, o servidor manterá o canal durante um período mínimo de tempo. Por outro lado, caso o canal tenha permanecido ocioso durante muito tempo, pode ser que o servidor não mais o esteja utilizando. Nesse caso, o invocador transita para o estado denominado `Probing`. Neste estado,

```
int LIM = 10;
MethodSet_Impl[] objs = new MethodSet_Impl[LIM];
for(int i = 0; i < LIM; i++)
    objs[i] = new MethodSet_Impl();
for(int i = 0; i < LIM; i++)
    RmeNaming.bind(args[0]+i, objs[i]);
for(int i = 0; i < LIM; i++)
    objs[i].activate();
```

Figura 4.4: Instanciação e ativação de vários objetos remotos.

o invocador utiliza as mensagens do tipo *ping* para verificar se o canal ainda pode ser utilizado. Em caso afirmativo, seu estado retorna para `Connected`, doutro modo o estado corrente passa a ser `Disconnected` e o estabelecimento de uma nova conexão faz-se necessário.

4.3.4 Ativação de Objetos Remotos

Em RME, para que um objeto remoto torne disponíveis os serviços que ele possui, são necessários dois passos principais. O primeiro deles é a sua ativação, e o segundo consiste no registro do objeto em uma agência de localização. Antes dessas etapas, contudo, é necessário que o objeto remoto seja instanciado. Durante a instanciação do objeto remoto ele é associado a um número de porta e ao endereço IP da máquina em que está sendo criado. Além disso, o objeto recebe um identificador que o caracterizará durante todo o seu ciclo de vida.

O componente `RmeActivator`, que implementa a classe `arcademis.Activator` é responsável por criar as estruturas de dados responsáveis pelo processamento de chamadas remotas e por inicializar a *thread* na qual é executado o receptor de invocações remotas. Tais estruturas englobam um `Dispatcher` e um *skeleton* a ele associado, além de um receptor de conexões, instância da classe `arcademis.BlockingAcceptor` A ativação de um objeto acontece quando o método `activate` é invocado sobre ele. Tal método é definido na interface `arcademis.Active`, que a classe `arcademis.RemoteObject` implementa, e que, por conseguinte, todo objeto remoto deve implementar.

Assim como em Java RMI, em RME é criada uma *thread* separada para a ativação e execução de cada objeto remoto. Dessa forma, o trecho de código visto na Figura 4.4 pode ser utilizado para instanciar 10 objetos remotos, registrá-los junto ao servidor de nomes e ativá-los logo em seguida. A fim de evitar que dois objetos remotos diferentes tentem utilizar a mesma porta para receber chamadas, referências remotas sempre são criadas com números de porta crescentes. Tais referências são criadas durante a instanciação de objetos do tipo `RmeRemoteObject`. Esta classe possui um atributo estático para armazenar o último valor de porta criado. Tal atributo é utilizado para garantir que novos valores sempre sejam criados a partir de um incremento do último valor de porta utilizado.

```
1:  RmeStream str1 = new RmeStream();
2:  str1.write((int)10);
3:  str1.write(true);
4:  str1.write('a');
5:  str1.write("Object of String type");
6:  RmeStream str2 = new RmeStream(str1.getBytes());
7:  int i = str2.readInt();
8:  boolean b = str2.readBoolean();
9:  char c = str2.readChar();
10: String s = (String)str2.readObject();
```

Figura 4.5: Exemplo de serialização de objetos e tipos primitivos.

4.3.5 O Mecanismo de Serialização Adotado

Sendo RME um sistema derivado de Arcademis, o mecanismo de serialização adotado nesta plataforma é uma extensão do modelo definido por aquele arcabouço. Conforme descrito na Seção 3.1.4, cada classe é responsável pela definição do algoritmo de serialização utilizado para transformar em seqüências de *bytes* as suas instâncias. Tal algoritmo é implementado no corpo do método `marshal` que cada objeto serializável deve fornecer. A fim de facilitar a tarefa do programador, o pacote RME define a classe `RmeStream`, que é uma extensão da classe `arcademis.Stream`.

Cada objeto do tipo `RmeStream` encapsula um arranjo de *bytes* sobre o qual seus métodos escrevem e lêem informações. Na linguagem Java, arranjos possuem capacidade fixa e limitada. Assim, sempre que a capacidade do *buffer* encapsulado por um `RmeStream` se esgota, uma realocação faz-se necessária: um novo arranjo com o dobro da capacidade da estrutura inicial é criado e, em seguida, procede-se à cópia do repositório original para seu substituto. Um exemplo de utilização de objetos do tipo `RmeStream` pode ser visto na Figura 4.5.

A serialização de tipos primitivos é trivial: tais tipos são decompostos nos *bytes* que os constituem e esses *bytes* são gravados em um `RmeStream`. Nenhum tipo de informação é adicionada como sufixo ou prefixo à seqüência de *bytes* que representa um tipo primitivo serializado. Dessa forma, após a execução da linha 2, na Figura 4.5, o objeto `str1` conterá quatro *bytes*, exatamente o tamanho de um inteiro na linguagem Java. *Bytes* são manipulados em `RmeStream` segundo o padrão adotado em filas: o primeiro *byte* gravado é o primeiro a ser lido.

Em RME, parâmetros de invocações que são objetos remotos são passados por referência, e não por valor, como acontece com todos os outros tipos de objetos. Assim, quando necessário serializar um objeto remoto, o *stub* gerado por aquele objeto é serializado, em vez do próprio objeto. Esta também é a estratégia de passagem de parâmetros de invocações remotas adotada por Java RMI, e permite que sejam realizadas chamadas remotas sobre objetos distribuídos recuperados de *streams*. Este tipo de chamada, conhecida por *call back*, apenas pode ser executada em aplicações implementadas sobre `Server_RME`, pois a versão `Client_RME` não possui recursos para comportar-se como servidora de requisições.

1: *Object*	→	*Type* | `NULL_REF`
2: *Type*	→	*Marsh* | *MarshA* | *String* | *StringA* | *PrimitiveA*
3: *Marsh*	→	*MarshType* `{Marsh.getClass().getName()}` `{Marsh.marshal()}`
4: *MarshType*	→	*Exception* | *NonExp*
5: *Exception*	→	`EXCEPTION_OBJECT` `{Exception.getMessage()}`
6: *NonExp*	→	`MARSHALABLE`
7: *MarsA*	→	`MARSHALABLE_ARRAY` `{(short)length}` `{Object[length]}`
8: *String*	→	`STRING` `{(short)length}` `{byte[length]}`
9: *StringA*	→	`STRING_ARRAY` `{(short)length}` `{String[length]}`
10: *PrimitiveA*	→	*BooleanA* | *ByteA* | *CharA* | *ShortA* | *IntA* | *LongA*
11: *BooleanA*	→	`BOOLEAN_ARRAY` `{(short)length}` `{boolean[length]}`
12: *ByteA*	→	`BYTE_ARRAY` `{(short)length}` `{byte[length]}`
13: *CharA*	→	`CHAR_ARRAY` `{(short)length}` `{char[length]}`
14: *ShortA*	→	`SHORT_ARRAY` `{(short)length}` `{short[length]}`
15: *IntA*	→	`INT_ARRAY` `{(short)length}` `{int[length]}`
16: *LongA*	→	`LONG_ARRAY` `{(short)length}` `{long[LongA]}`

Tabela 4.1: Protocolo de serialização adotado em RME.

O Protocolo de Serialização

Conforme dito anteriormente, a serialização de tipos primitivos é um processo simples que sempre produz arranjos de *bytes* cujo tamanho é igual ao tamanho do tipo definido pela linguagem Java. Assim, valores booleanos podem ser codificados com um *byte*, caracteres e valores do tipo `short` são codificados com dois *bytes*, inteiros com quatro e inteiros longos (`long`) com oito *bytes*. Por outro lado, a serialização de tipos compostos, tais como objetos e arranjos de objetos, ou mesmo arranjos de tipos primitivos, não é um processo tão simples, e pressupõe a existência de um protocolo de serialização, o qual foi definido na classe `RmeStream` e aparece representado na Tabela 4.1.

Com o intuito de tornar o processo de serialização mais eficiente, a codificação de *strings* é tratada como um caso particular, uma vez que o tipo `String` é extremamente utilizado em aplicações em geral. Um `String` serializado consiste de um arranjo de *bytes* no qual os dois primeiros elementos representam o seu tamanho e os elementos restantes o seu conteúdo. Esta é a mesma representação adotada para arranjos de tipos primitivos serializados. Além de permitir a serialização de *strings*, o protocolo de serialização de RME também contém rotinas para a serialização de subclasses de `java.lang.Exception`. Isto permite que servidores tenham como notificar clientes sobre a ocorrência de situações excepcionais durante o processamento de chamadas remotas. Caso a classe de exceção contenha outros atributos que devam ser serializados, então ela precisa implementar a interface `Marshalable`.

A classe `RmeStream` não predefine rotinas para a serialização de estruturas como arranjos de duas ou mais dimensões. A fim de contornar tal limitação, pode-se estender `RmeStream` e adicionar os métodos necessários, ou pode-se encapsular a estrutura que se deseja serializar em

```
import arcademis.Marshalable; import arcademis.MarshalException;

public class ExSerialization implements Marshalable {
    private int i = 0;
    private String s = null;

    public ExSerialization() {}

    public ExSerialization(int i, String s) {
     this.s = s;
     this.i = i;
    }
    public void marshal(Stream b) throws MarshalException {
        b.write(i);
        b.write(s);
    }
    public void unmarshal(Stream b) throws MarshalException {
        i = b.readInt();
        s = (String)b.readObject();
    }
}
```

Figura 4.6: Exemplo de objeto serializável em RME.

um objeto do tipo `Marshalable`. Existem outros tipos de dados que não podem ser serializados pela classe `RmeStream`. Por exemplo, dentre todas as classes de Java, RME somente predefine rotinas de serialização para as classes `String` e `Exception`. Caso seja necessário serializar outros tipos compostos, estes precisam implementar a interface `Marshalable` e, por conseguinte, os métodos `marshal()` e `unmarshal()`.

Todos as classes que podem ser serializadas em RME devem implementar um construtor sem parâmetros, caso possuam outros construtores. Isto se deve ao fato de o método `readObject`, de `RmeStream` necessitar deste construtor para instanciar objetos uma vez que seu tipo houver sido lido da seqüência de *bytes*. A implementação do método `marshal()` é utilizada como última etapa do processo de serialização. Conforme pode ser visto na linha 3 da Tabela 4.1, esse método é invocado uma vez que todos os prefixos que qualificam o objeto sendo serializado já foram gravados em um `RmeStream`. Os métodos `marshal()` e `unmarshal()` devem conter, respectivamente, os procedimentos necessários para gerar uma seqüência de *bytes* que descreve o objeto serializável e as rotinas necessárias para preencher os atributos do objeto a partir da leitura de uma cadeia de *bytes*. Um exemplo de classe que implementa a interface `Marshalable` é apresentado na Figura 4.6.

4.3.6 O Protocolo de Comunicação

O protocolo JRMP [Mic03], utilizado pela plataforma Java RMI para o transporte de dados, é pouco adequado ao ambiente no qual executam computadores móveis [CKR00]. São duas, as causas principais de tal inadequação. Inicialmente cita-se o fato de JRMP utilizar mensagens de tamanho considerável em termos de número de *bytes*. A segunda destas causas é o fato do protocolo utilizar diversas mensagens, entre requisições e confirmações, para permitir que objetos distribuídos se comuniquem. Por exemplo, a localização de um objeto distribuído e a realização

de uma chamada remota sobre um de seus métodos, em RMI, exigiria entre seis e oito trocas de mensagens, ao passo que dessas, somente duas mensagens contêm informação pertinente à invocação remota propriamente dita [CKR00]. Dispositivos móveis utilizam conexões de capacidade limitada e normalmente intermitentes, e, por isso, requerem um protocolo de comunicação mais otimizado.

A plataforma RME utiliza um protocolo de comunicação denominado RMEP, ou *RME Protocol*, o qual foi desenvolvido com o objetivo de utilizar um número pequeno de mensagens para permitir a comunicação entre objetos distribuídos. Esse protocolo utiliza quatro tipos de mensagens diferentes, as quais foram nomeadas `call`, `return`, `ping` e `ack`. Mensagens do tipo `call` caracterizam invocações remotas de métodos e são respondidas por mensagens do tipo `return`, as quais contêm valores que resultam da execução de algum método remoto. A Figura 3.11 mostrada na Seção 3.1.5 contém a implementação da classe `CallMsg` que define mensagens do tipo `call`. As mensagens do tipo `ping` e `ack` são utilizadas para verificar se existem servidores de métodos remotos ativos em um dado endereço. Por exemplo, conforme discutido na Seção 4.3.3, a implementação do *stub* utiliza uma mensagem do tipo `ping` para verificar se o canal de comunicação criado durante a realização de chamadas remotas anteriores pode ser aproveitado.

A Tabela 4.2 contém uma gramática que define o formato das mensagens utilizadas por RMEP. Nesta gramática, os símbolos `call`, `ping`, `return` e `ack` denotam constantes que caracterizam o tipo da mensagem, as quais foram separadas em duas categorias. A primeira delas é composta por mensagens do tipo `call` e `ping`. Tais mensagens exigem uma resposta, de acordo com o protocolo RMEP, e, por isso, incluem um endereço para retorno. O símbolo `epid` representa a seqüência de *bytes* obtida da serialização de um objeto do tipo `HostPortEpid`, o qual contém o nome do *host* e o número da porta que definem o endereço para o qual a mensagem de resposta deve ser enviada. Identificadores de chamadas remotas e identificadores de objetos são representados na gramática dada, respectivamente, pelos símbolos $call_{id}$ e obj_{id}. Para a manipulação de tais identificadores, a plataforma RME define a classe `RmeIdentifier`, representada, na Tabela 4.2 pelo símbolo `id`. Finalmente, os argumentos de uma chamada remota e o valor de retorno da mesma são representados pelo símbolo `Stream`, que denota uma seqüência de *bytes* obtida da serialização de objetos e de tipos primitivos, segundo o protocolo apresentado na Seção 4.3.5.

4.3.7 O Serviço de Localização de Nomes

Conforme discutido na Seção 4.2.1, RME baseia-se em um modelo conhecido como arquitetura de serviço, o qual é composto por três tipos principais de entidades: clientes, fornecedores de serviços e agências de localização. Conforme tratado na Seção 3.1.12, Arcademis não força o desenvolvedor de *middleware* a adotar uma interface específica para a agência de localização, embora disponibilize alguns componentes com este fim. RME, contudo, não utiliza nenhuma das interfaces fornecidas por Arcademis, pois optou-se por fornecer para este *middleware* a mesma

$$
\begin{array}{rcl}
message & \rightarrow & header\ version\ msg_{type} \\
header & \rightarrow & \texttt{0x52 0x4d 0x45 0x50} \\
version & \rightarrow & \texttt{0x01} \\
msg_{type} & \rightarrow & msg_{req} \mid msg_{ack} \\
msg_{req} & \rightarrow & \texttt{epid}\ req_{type} \\
req_{type} & \rightarrow & \texttt{call}\ call_{id}\ obj_{id}\ op\ args \\
 & \mid & \texttt{ping} \\
call_{id} & \rightarrow & \texttt{id} \\
obj_{id} & \rightarrow & \texttt{id} \\
op & \rightarrow & \texttt{int} \\
args & \rightarrow & \texttt{Stream} \\
msg_{ack} & \rightarrow & \texttt{return}\ call_{id}\ ret_{val} \\
 & \mid & \texttt{ack} \\
ret_{val} & \rightarrow & \texttt{Stream}
\end{array}
$$

Tabela 4.2: Formato de mensagens do protocolo RMEP.

sintaxe de acesso à agência de localização que a presente em Java RMI. Em Java RMI, a agência de localização é implementada por meio de uma classe denominada `Naming` [Mic03]. Na plataforma RME foi definida a classe `RmeNaming`. O serviço fornecido por RME disponibiliza ao usuário as mesmas funcionalidades encontradas em Java RMI, tais como a associação de nomes a *stubs* e a busca de objetos remotos a partir dos nomes a que foram associados.

Assim como em Java RMI, também em RME o serviço de localização é controlado por um objeto remoto, o *servidor de nomes*. Em RME tal objeto remoto recebe conexões na porta 1101. A classe `RmeNaming` encapsula o acesso ao servidor de nomes, disponibilizando aos desenvolvedores de aplicações uma sintaxe mais simples do que a que seria necessária caso referências remotas fossem utilizadas diretamente para permitir o acesso àquele diretório. A implementação da classe `RmeNaming` pode ser descrita como uma tabela em que estão associadas instâncias de *stubs* a nomes que as representam e que funcionam como chaves de pesquisa.

Existem duas implementações de `RmeNaming`: uma adequada a aplicações clientes e outra adequada a aplicações servidoras. Tal diferença se explica porque clientes não precisam registrar-se no diretório de nomes, de modo que, para eles, os métodos necessários a este registro são supérfluos. A versão cliente de `RmeNaming` possui somente dois métodos: `lookup` e `list`, sendo a função da primeira operação realizar consultas sobre uma dada agência de localização e a função do segundo método retornar uma lista com todos os nomes presentes naquela agência. A versão de `RmeNaming` implementada para aplicações servidoras fornece mais três rotinas: `bind`, `rebind` e `unbind`. Destes métodos, os dois primeiros realizam a associação, na agência de localização, entre a implementação de um objeto remoto e o nome que o irá qualificar. A diferença entre eles é que `bind` causa o levantamento de uma exceção caso o nome especificado já esteja em uso. O último dos métodos, denominado `unbind`, destrói a associação existente entre um nome e o objeto a ele associado. As duas interfaces podem ser vistas na Figura 4.7.

```
package rme.naming;
import arcademis.*;
import arcademis.server.*;
public interface ServerNamingService {
  public void bind(String n, Stub s)
  throws AlreadyBoundException,
  ArcademisException;

  public void unbind(String name)
  throws NotBoundException,
  ArcademisException;

  public void rebind(String n, Stub s)
  throws ArcademisException;
}
```

```
package rme.naming;
import arcademis.*;
import arcademis.server.*;
public interface ClientNamingService{
  public Remote lookup(String name)
  throws NotBoundException,
  ArcademisException;

  public String[] list()
  throws ArcademisException;
}
```

Figura 4.7: Interface para registro e localização de nomes de RME.

Consultas à agência de localização sempre verificam se objetos remotos estão recebendo chamadas antes de retornar uma referência para eles, quando necessário responder alguma requisição. Esse procedimento evita que referências para servidores não ativos sejam retornadas para clientes. Este teste também pode ocorrer durante a execução do método `bind`. Quando uma tentativa de associar um *stub* a um nome já existente acontecer, a agência de localização verifica se o objeto remoto representado por aquele nome está ativo. Caso tal fato não se verifique, a antiga associação é removida e uma nova relação é, a seguir, adicionada ao diretório de nomes.

4.3.8 A Configuração do ORB

A plataforma RME define duas configurações para o ORB: uma adequada para aplicações clientes (`Client_RME`) e outra voltada para aplicações servidoras (`Server_RME`). A configuração servidora define duas fábricas a mais que a configuração cliente, que por sua vez cria e associa ao ORB sete fábricas de objetos. Cada uma destas fábricas é apresentada na Tabela 4.3. Dentre aquelas fábricas, algumas são usadas somente por `Server_RME`, como `AcceptorFc` e `ConnectionServerFc`. As fábricas podem ser utilizadas via a classe `arcademis.OrbAccessor`, descrita na Seção 3.2, ou podem ser obtidas diretamente a partir do ORB.

A utilização de fábricas para a criação dos componentes que fazem parte da plataforma RME tem o objetivo de facilitar a reconfiguração do sistema. Para que tal objetivo possa ser alcançado, alguma disciplina de programação faz-se necessária: os desenvolvedores não devem criar objetos diretamente. A instanciação de componentes do *middleware* deve ser feita por meio de fábricas de objetos. A linguagem Java não dispõe de um mecanismo que previna a instanciação de objetos sem adicionar muitas restrições à classe que os define. Por exemplo, a fim de impedir que um componente fosse instanciado diretamente, poder-se-ia declarar os métodos construtores da classe que o implementa como privados. Entretanto, classes cujos construtores são todos privados não podem ser estendidas em Java, o que poderia vir a ser uma restrição por demais severa. Assim, a reconfigurabilidade de qualquer instância de Arcademis depende da disciplina

Fábrica	Interface implementada	Componente criado	Ocorrência
`RmeAcceptorFc`	`AcceptorFc`	`BlockingAcceptor`	s
`RmeActivatorFc`	`ActivatorFc`	`RmeActivator`	s
`RmeChannelFc`	`ChannelFc`	`TcpSocketChannel`	s/c
`RmeConnectorFc`	`ConnectorFc`	`SynchronousConnector`	s/c
`RmeConServerFc`	`ConnectionServerFc`	`TcpSocketServer`	s
`RmeDispatcherFc`	`DispatcherFc`	*dispatcher*	s
`RmeEpidFc`	`EpidFc`	`HostPortEpid`	s/c
`RmeIdentifierFc`	`IdentifierFc`	`RmeIdentifier`	s/c
`RmeInvokerFc`	`InvokerFc`	*invokers*	s/c
`RmeMessageFc`	`MessageFc`	*messages*	s/c
`RmeRemoteRefFc`	`RemoteRefFc`	`RemoteReference`	s/c
`RmeProtocolFc`	`ProtocolFc`	`RmeProtocol`	s/c
`RmeServiceHandlerFc`	`ServiceHandlerFc`	*service handlers*	s/c
`RmeStreamFc`	`StreamFc`	`RmeStream`	s/c

Tabela 4.3: Fábricas de objetos utilizadas em RME.

adotada pelos programadores que utilizam o arcabouço.

4.3.9 Implementação de Pontos de Localização e Identificadores

Identificadores Remotos

Identificadores são componentes utilizados em RME para caracterizar de forma única referências remotas e invocações de métodos. Tais componentes são instâncias da classe `arademis.concreteComponents.HostTimeCountId`. Esta classe utiliza uma técnica similar à que é utilizada por Java RMI para tentar garantir a unicidade de suas instâncias. Tal técnica não assegura que dois identificadores distintos são sempre diferentes, porém a probabilidade de que eles sejam iguais é extremamente baixa. Assim como Java RMI, a implementação de identificadores fornecida por RME utiliza três atributos diferentes para individualizar cada uma de suas instâncias. Tais atributos estão relacionados a seguir:

vmd (*Virtual Machine Discriminator*): esse valor é utilizado para diferenciar identificadores gerados por máquinas virtuais diferentes. Tal valor é produzido aleatoriamente durante a inicialização do *middleware* e é o mesmo em todos os identificadores criados na mesma máquina virtual. A chance que dois valores criados em máquinas virtuais diferentes coincidam é de $1/(2^{32}-1)$.

td (*Time Discriminator*): esse valor diferencia identificadores gerados em instantes de tempo diferentes. A precisão de tempo adotada é de milisegundos, ou seja, o valor armazenado em **td** é o número de milisegundos que se passaram desde o primeiro dia de Janeiro de 1970 até a data marcada no relógio do computador onde o objeto foi criado.

count : esse atributo é utilizado para diferenciar identificadores criados com o mesmo valor **td** na mesma máquina virtual. Uma vez que os processadores modernos são capazes de criar centenas de objetos em um intervalo menor que 1 milisegundo, a possibilidade de identificadores diferentes serem criados com o mesmo valor para **td** não pode ser desprezada. A classe `HostTimeCountId` possui um atributo estático que representa o número de vezes que foi instanciada. Tal atributo é usado para definir o valor **count** para cada objeto criado.

A principal função de identificadores em RME é assegurar que um objeto remoto processe somente as mensagens originadas por *stubs* que foram criados por ele. Dessa forma, não é possível que programas criem *stubs* localmente e os preencham com a localização de um objeto remoto de modo a utilizá-los como se eles houvessem sido criados pelo servidor de nomes em resposta a uma operação de pesquisa. Tal não é factível porque mensagens são acrescidas com o identificador do *stub* que as originou. O *middleware*, contudo, não protege as mensagens contra cópia ou falsificação, de forma que é possível que uma aplicação intercepte as mensagens enviadas e copie o cabeçalho das mesmas, a fim de poder se comunicar com a implementação do objeto remoto para o qual tais mensagens eram endereçadas.

Pontos de Localização

Conforme discutido na Seção 3.1.2, endereços de entidades remotas, em Arcademis, são representados pela interface `Epid`. A implementação de pontos de localização fornecida por RME baseia-se no modelo adotado pelo protocolo TCP/IP para a localização de recursos distribuídos. Cada endereço remoto é definido por um nome de *host*, ou endereço IP, e por um número de porta. Enquanto objetos remotos sempre recebem conexões em alguma porta cujo número identificador se encontra entre 1200 e 30000, o serviço de localização recebe conexões na porta 1101.

4.3.10 Despacho de Invocações Remotas

O componente `Dispatcher` utilizado em RME recebe invocações remotas do *request receiver* e as repassa diretamente para o *skeleton* a que está associado. Além de servir como um elemento de ligação entre o *skeleton* e a camada de transporte, o *dispatcher* possui outras duas funções: verificar se as chamadas remotas recebidas são provenientes de um *stub* válido e tratar a ocorrência de exceções durante o processamento das mesmas. A implementação da classe `RmeDispatcher` é mostrada na Figura 4.8.

Para determinar se as chamadas remotas recebidas foram criadas por um *stub* válido, isto é, por um *stub* que possui o mesmo identificador que a implementação do objeto remoto, o *dispatcher* compara os identificadores presentes nas requisições com o identificador do *skeleton* que ele encapsula. Tal verificação ocorre na classe `RmeDispatcher`, cuja implementação pode ser vista na Figura 4.8. Em relação ao tratamento de situações excepcionais, a implementação

```
public class RmeDispatcher implements Dispatcher {
  Skeleton skeleton = null;
    public void setSkeleton(Skeleton skeleton) {
    this.skeleton = skeleton;
  }
  public Stream dispatch(RemoteCall call) {
    Stream returnValue = null;
    try {
      if(!comesFromValidStub((RmeRemoteCall)call)) {
        IncompatibleStubException ise = new IncompatibleStubException();
        returnValue.write((Exception)ise);
      } else {
        try {
          returnValue = this.skeleton.dispatch(call);
        } catch (Exception e) {
            // the exception must be reported to the client application.
            returnValue = OrbAccessor.getStream();
            returnValue.write(e);
        }
      }
    } catch (MarshalException me) {}
    return returnValue;
  }
  public boolean comesFromValidStub(RmeRemoteCall rCall) {
    RemoteObject remoteObject = skeleton.getRemoteObject();
    RemoteReference ref = remoteObject.getRef();
    Identifier id = ref.getIdentifier();
    return id.equals(rCall.getTargetObjectIdentifier());
  }
}
```

Figura 4.8: Código da classe `RmeDispatcher`.

do *dispatcher* é responsável por capturar, serializar e enviar de volta para o cliente todas as exceções disparadas no corpo do *skeleton*.

O `Dispatcher` recebe invocações remotas por meio da classe `RmeRequestReceiver`. Instâncias desta classe são criadas para cada conexão estabelecida com o servidor e são responsáveis por tratar mensagens do tipo *ping* e *call*. A implementação desta classe pode ser vista na Figura 4.9. Em RME, *request receivers* são executados em uma *thread* separada da aplicação principal, de modo a poderem tratar em separado cada nova conexão criada com o servidor. Estas conexões, contudo, não são mantidas abertas para sempre. Depois de um certo intervalo de tempo o canal é fechado. Na implementação mostrada na Figura 4.9 o intervalo de tempo mencionado é de 2 segundos.

4.3.11 Semântica de Recebimento de Chamadas adotada em RME

Um serviço de invocação remota de métodos pode utilizar vários tipos diferentes de semântica relacionadas ao processamento de chamadas remotas. Estas semânticas caracterizam os diversos níveis de garantia que servidores fornecem para os clientes acerca do processamento de invocações de métodos. As abordagens mais comuns foram apresentados na Seção 3.1.6. Por razões de eficiência, a plataforma RME adota um modelo similar à política conhecida como *best effort*, porém uma outra política, mais robusta foi implementada e, graças à utilização de fábricas de

```
public class RmeRequestReceiver extends RequestReceiver
implements RmeConstants, Runnable, ProtocolHandler {

  protected Dispatcher dispatcher = null;
  private static final int TOLERANCE = 2000; // 2 seconds

  public void open(Channel ch) {
    ch.setConnectionTimeout(TOLERANCE);
    super.protocol.setChannel(ch);
    Thread t = new Thread(this);
    t.start();
  }

  public void run() {
    boolean isOperant = true;
    while(isOperant) {
      try {
        Message msg = super.protocol.recv();
        if(msg instanceof CallMsg) {
          RmeRemoteCall remoteCall = (RmeRemoteCall)((CallMsg)msg).getRemoteCall();
          Stream returnValue = dispatcher.dispatch(remoteCall);
          this.sendReturnValue(remoteCall.getCallIdentifier(),
          returnValue, super.protocol);
        } else if (msg instanceof PingMsg) {
          AckMsg ack = (AckMsg)OrbAccessor.getMessage(RmeConstants.PING_ACKNOWLEDGE);
          super.protocol.send(ack);
        } else {throw new ProtocolException();}
      } catch (Exception e) {
        try {super.protocol.getChannel().close();}
      } catch(NetworkException ne) {isOperant = false;}
    }
  }

  private void sendReturnValue(Identifier callId, Stream returnStr, Protocol rmep) {
    // create a ResponseReceiver and invoke its open method.
    // ... not shown here due to space constraints
  }
}
```

Figura 4.9: Código da classe `RmeRequestReceiver`.

objetos, esta pode ser adotada sem adicionar alterações ao código da plataforma, a não ser no código da fábrica responsável pela criação de processadores de serviço.

Em RME, toda chamada remota executada com sucesso pressupõe uma mensagem de resposta, ainda que o método invocado não retorne qualquer valor. A mensagem de resposta permite que aplicações clientes tenham como saber se uma chamada remota foi executada. Diante da ausência de uma mensagem de retorno, dois cenários diferentes são possíveis. No primeiro, a mensagem contendo a invocação remota perdeu-se antes de ser recebida pela implementação do objeto remoto e, por isso, não chegou a ser processada por ele. No segundo cenário, a chamada foi recebida e processada pelo fornecedor de serviço, porém uma falha de comunicação originou a perda da mensagem de resposta. Caso o cliente não venha a receber a confirmação de execução de uma invocação remota, ele não tem como saber se esta foi processada pelo servidor ou não.

Caso seja necessário prover às aplicações clientes maiores níveis de garantia acerca da execução de chamadas remotas, RME fornece versões das interfaces `arcademis.RequestSender` e `arcademis.RequestReceiver` que implementam a semântica *at most once*. Conforme discutido na Seção 3.1.6, essa política é adequada para aplicações em que métodos remotos podem modificar o estado dos servidores, pois ela visa garantir que chamadas remotas sejam processadas no máximo uma vez. Um sistema distribuído para controle de inventário é um exemplo em que a semântica *at most once* é interessante. Considera-se, neste caso, diversas aplicações responsáveis pela efetivação de vendas e um servidor que mantém o estoque de produtos de uma loja. Quando algum item é vendido, a aplicação cliente informa o servidor sobre a venda via uma chamada remota de método. Esta chamada faz com que o banco de dados do servidor seja atualizado, decrementando-se uma ocorrência do produto vendido. Assim, caso uma chamada remota seja processada pelo servidor mais de uma vez, o banco de dados ficará em um estado inconsistente com o estoque real de mercadorias. A semântica *at most once* garante que este tipo de erro não acontecerá, ou que acontecerá apenas em situações tão raras que podem ser desconsideradas.

Na solução adotada por RME o receptor de requisições mantém uma tabela que associa a cada cliente, identificado pelo seu endereço remoto, uma lista dos últimos identificadores de chamadas enviados por ele. Tal lista é implementada como uma fila circular [CLRS02], sendo a sua capacidade definida em tempo de compilação. Identificadores são armazenados na ordem em que são recebidos, e os identificadores mais antigos que o mais antigo identificador armazenado são descartados. O emissor de requisições envia periodicamente a mesma mensagem até que uma resposta seja obtida, ou um limite de tentativas seja atingido, quando então a ocorrência de erro é reportada para a aplicação.

A Figura 4.10 mostra as máquinas de estado do *request sender* (a) e do *request receiver* (b) que caracterizam a implementação da semântica *at-most-once* em RME. De acordo com este esquema, o *request sender*, após efetuar o envio de uma requisição, inicia um contador de tempo, e, não recebendo qualquer resposta do servidor em um dado intervalo, torna a enviar a requisição. Caso o cliente não receba qualquer confirmação de processamento por parte do

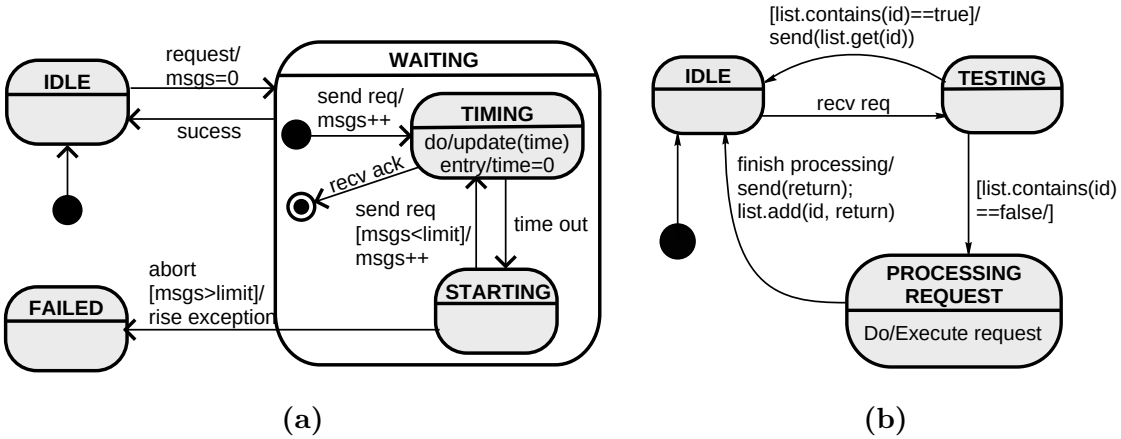

Figura 4.10: Máquinas de estado que caracterizam a semântica *at-most-once*.

servidor após um certo número de tentativas de comunicação, ele assume que o objeto remoto está inalcançável, sendo uma exceção disparada em seguida. Do lado do servidor, sempre que o *request-receiver* recebe uma invocação remota, ele verifica se o identificador da mesma está presente na lista de chamadas processadas. A chamada é repassada para o `Dispatcher` apenas se tal identificador não for encontrado.

O mecanismo implementado visa assegurar que uma mensagem seja processada pelo servidor no máximo uma vez. Contudo, existem probabilidades mínimas de que uma chamada venha a ser tratada mais vezes. Tal fato pode acontecer devido a atrasos no canal de comunicação utilizado entre as partes servidora e cliente em conjunto com o fato da implementação da semântica *at most once* provida por RME utilizar uma lista finita para armazenar os identificadores de mensagens. A fim de ilustrar uma possível falha, pode-se supor um cenário hipotético em que é utilizado um *buffer* de quatro posições para armazenar identificadores. Supõem-se também que as mensagens sejam rotuladas por letras do alfabeto, de modo que à primeira mensagem corresponda o caracter A, à segunda B e assim por diante.

No cenário exposto anteriormente, suponha que um cliente tenha enviado cinco mensagens para o servidor, rotuladas de A até E, porém, devido a atrasos na rede, a última mensagem (E) tenha sido recebida na frente de todas as outras. Devido àlguma falha na camada de transporte, suponha que a resposta relativa à mensagem E tenha se perdido antes de alcançar a aplicação cliente e que, nesse ínterim, todas as mensagens, de A até D tenham sido recebidas pelo servidor, e que seus identificadores agora preencham completamente o *buffer*, que por hipótese possui somente quatro posições. Ao constatar a ausência da mensagem de resposta, o cliente reenviará a mensagem E. Uma vez que esta mensagem possui um identificador mais novo que todos os identificadores armazenados no *buffer*, ela será processada pelo servidor, muito embora já tenha sido recebida anteriormente. Como é possível notar, embora pequena, a probabilidade de que ocorra uma falha de natureza semântica é possível na implementação da política *at most once* fornecida por RME.

Outro problema que decorre da implementação da política *at most once* é a possibilidade de mensagens ainda não processadas serem descartadas antes de serem tratadas. Tomando por base o exemplo descrito anteriormente, caso o cliente execute cinco chamadas remotas, rotuladas de A até E para o servidor remoto, então se a primeira mensagem se atrasar o suficiente para ser precedida por todas as outras, então ela será descartada logo que recebida, pois no *buffer* haverá quatro identificadores mais novos que A. A fim de diminuir a possibilidade deste tipo de evento, mensagens somente são descartadas caso sejam mais antigas que todas as outras armazenadas na lista de chamadas recebidas.

A Tabela 4.4 mostra uma comparação entre o desempenho das duas políticas implementadas: *at most once* e *best effort*. Foram comparados 13 métodos remotos, cujas definições são mostradas na Figura 4.27. Tais métodos diferem pelo valor de retorno e pelo tipo dos parâmetros. Na tabela pode ser visto o tamanho de cada requisição, em *bytes*, que abarca o espaço ocupado pelo cabeçalho da mensagem remota e pelos argumentos serializados do método. Também são mostrados os tamanhos das mensagens de resposta, em *bytes*, dado pelo cabeçalho da mensagem de retorno e pelo valor retornado, quando este existe. Para a realização de tais testes, as máquinas virtuais cliente e servidora foram executadas em um mesmo computador, um Pentium 4 de 2.0GHz e 512MB de RAM. Cada um dos 13 métodos apresentados na interface da Figura 4.27 foi executado 10000 vezes com sucesso, ou seja, não foram forçadas falhas na execução de qualquer dos métodos. A média de 10 destas seqüências de execução constituem os resultados apresentado na tabela. Pela análise das informações mostradas, conclui-se que a semântica *at-most-once* não adiciona atrasos substanciais ao tempo de processamento de invocações remotas. O impacto mais severo é em termos de espaço: o servidor precisa manter um *buffer* para cada cliente a ele conectado.

É importante destacar que a modificação da semântica de processamento de chamadas é uma tarefa extremamente simples devido ao projeto de RME, baseado em fábricas de objetos. Essencialmente, tal modificação pode ser levada a diante por meio da alteração de poucas linhas de código na fábrica de processadores de serviço, uma vez que os componentes encarregados da semântica das chamadas já se encontram implementados. Basta modificar os comandos responsáveis pela criação do *request sender* e do *request receiver*. Tal fato é possível por que a semântica de invocações remotas não depende de outros componentes do *middleware* que não os processadores de serviço.

4.4 Versões de RME para Clientes e Servidores

Foram desenvolvidas duas versões de RME: uma versão para aplicações servidoras (`Server_RME`) e outra para aplicações clientes (`Client_RME`), que pode ser considerada um subconjunto da primeira versão citada. Os tamanhos dos pacotes cliente e servidor são, respectivamente, 37KB e 68KB. A versão cliente foi desenvolvida com o intuito de conter apenas os recursos impres-

Tipo dos Parâmetros	Tamanho da Requisição	**Tipo de Retorno**	Tamanho da Resposta	**Best Effort** *req/s*	**At Most Once** *req/s*
empty	200	short	70	4081,73	4019,98
empty	200	char	70	4177,98	4128,81
empty	200	int	72	4133,94	4085,30
empty	200	long	76	4142,50	4092,66
empty	200	String	81	4058,98	4009,62
empty	200	String[]	201	3348,03	3319,61
byte,short,char, int,long,String, String[]	351	void	69	3942,18	3869,97
byte[]	213	String	82	3917,99	3859,81
short[]	223	String	82	3798,42	3725.23
char[]	223	String	90	3851,33	3818,83
int[]	243	String	82	3671,30	3627,92
long[]	283	String	82	2891,57	2847,06
String[]	323	String	170	3226,50	3082,23

Tabela 4.4: Comparação de desempenho entre semânticas de chamada remota.

cindíveis para a realização de chamadas remotas. Assim, tal pacote não contém, por exemplo, implementações de classes tais como `RmeRemoteObject`, `RmeRequestReceiver`, `RmeResponse-Sender`, `RmeSkeleton` e `TcpServerSocket`.

Conforme tratado na Seção 4.2, a plataforma RME possui distribuições tanto para a edição J2SE quanto para a edição J2ME da linguagem Java. A distribuição voltada para J2SE apresenta as versões servidora e cliente de RME, porém a distribuição desenvolvida para J2ME apresenta somente a versão cliente. Não foi desenvolvida uma versão servidora para essa plataforma por razões pragmáticas: dificilmente uma aplicação que é executada em um aparelho celular poderia ser utilizada como um servidor de invocações remotas de métodos, pois tal função demanda recursos tais como capacidade de processamento e grande disponibilidade de memória, recursos estes que um telefone móvel dificilmente possuiria. Além disso, as bibliotecas que compõem a versão servidora de RME são cerca de duas vezes maiores que aquelas desenvolvidas para a versão cliente. O tamanho do pacote servidor é, assim, uma severa limitação, pois muitos dispositivos móveis simplesmente não possuem uma quantidade de memória suficiente para armazenar todos os arquivos necessários.

Uma vez que funcionalidades que pertencem ao domínio de servidores de chamadas remotas não são fornecidas pelo pacote `Client_RME`, não é possível utilizar, sobre a plataforma J2ME, aplicações baseadas em RME capazes de receber *call backs*. Call back é o nome atribuído a um método que é executado sobre um cliente pelo servidor em resposta a uma invocação remota originada pelo cliente. A título de exemplo, há, essencialmente, dois modos pelos quais um cliente pode ser notificado acerca da ocorrência de um evento em um *host* remoto. No primeiro

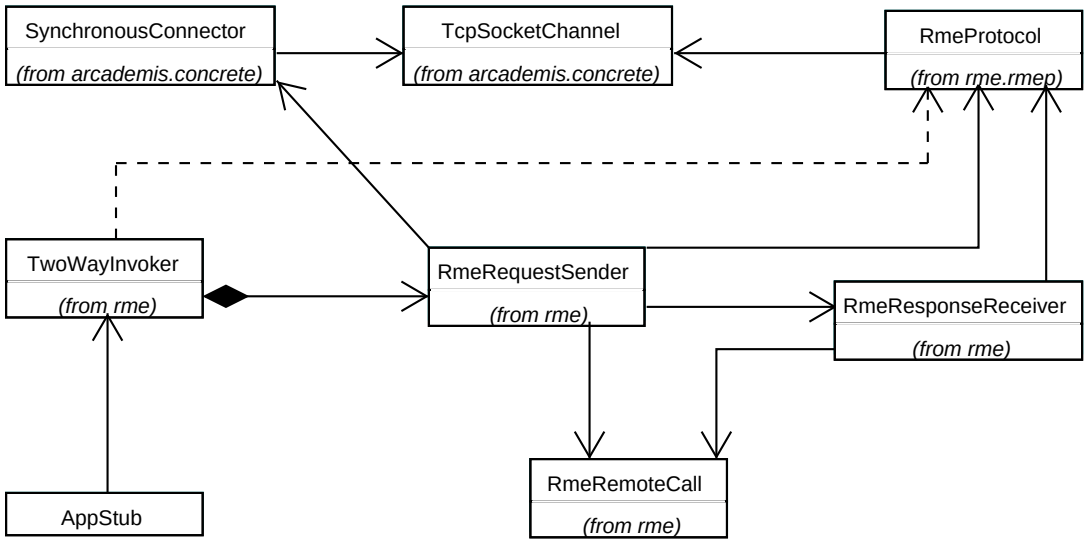

Figura 4.11: Principais classes usadas por clientes para enviar e receber requisições.

modo, o cliente continuamente verifica a ocorrência do evento em questão por meio de conexões a um servidor localizado naquele *host*. Na segunda forma, o cliente informa um servidor localizado no *host* remoto sobre o seu interesse no evento, e passa a aguardar um aviso de resposta. Tal aviso pode ser implementado como um *call back*. Com a versão cliente de RME, o mecanismo de *call back* não pode ser utilizado, pois a aplicação não tem como criar um servidor para esperar pela mensagem de retorno.

A implementação do serviço de localização também apresenta diferenças entre as versões cliente e servidora de RME. Conforme discutido na Seção 4.3.7, o acesso ao diretório de nomes se dá via a classe `RmeNaming`, que fornece ao desenvolvedor de aplicações cinco métodos: `lookup`, `bind`, `rebind`, `unbind` e `list`. Estes cinco métodos, contudo, estão presentes somente na versão servidora do *middleware*. A versão cliente do mesmo disponibiliza para as aplicações distribuídas somente os métodos `lookup` e `list`. A ausência dos demais métodos se justifica na medida em que eles são utilizados pela implementação de objetos remotos para que estas possam se registrar em uma agência de localização. Uma vez que essa capacidade de registro não é necessária para aplicações clientes, elas foram suprimidas.

A Figura 4.11 mostra como estão relacionadas as principais classes responsáveis pelo envio e recebimento de requisições remotas no lado cliente de uma aplicação distribuída. Nessa figura, a classe `AppStub` foi gerada automaticamente por `RmeC`. A Figura 4.12 mostra as principais classes envolvidas no processamento de chamadas remotas no lado do servidor. Algumas relações foram omitidas a fim de diminuir a complexidade do diagrama apresentado.

Os principais pacotes que constituem Arcademis e RME são mostrados na Figura 4.13. O pacote `arcademis.Concrete` contém a implementação de alguns componentes concretos que podem ser aproveitados em plataformas de *middleware* sem qualquer alteração e `arcademis.server`

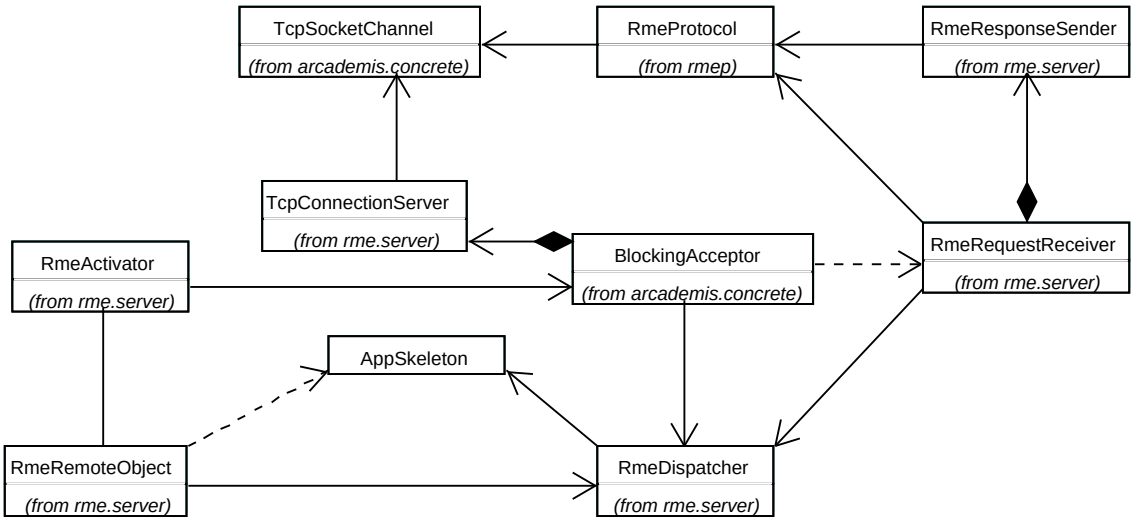

Figura 4.12: Principais classes usadas por servidores para enviar e receber requisições.

contém componentes do arcabouço exclusivos para aplicações servidoras. `Rme.server` contém classes da parte servidora de RME. `Rme.rmep` implementa o protocolo de comunicação do *middleware*. `Rme.extras` contém algumas classes que podem ser usadas no lugar de outras que normalmente são utilizadas, como a implementação da semântica *at-most-once*, por exemplo. RME possui ainda outros dois subpacotes: `rme.naming` e `rme.rmec`. O primeiro contém a implementação da agência de localização e o segundo implementa o gerador de *stubs* e *skeletons*.

4.5 O Gerador de *Stubs* e *Skeletons*

A codificação tanto de *stubs* quanto de *skeletons* é uma tarefa que pode ser facilmente automatizada por um gerador de código. Em Java RMI, *stubs* e *skeletons* são gerados por um compilador conhecido como `rmic` [Mic03]. Para RME foi implementada uma ferramenta análoga, denominada `rmec`, ou *Rme Compiler*. Tal ferramenta foi desenvolvida a partir de outro gerador automático de código, implementado para o projeto Ninja RMI [Wel03], e utiliza o mecanismo de reflexividade da linguagem Java para analisar a implementação de um objeto remoto a fim de produzir um *stub* e um *skeleton* para ele. A reflexividade evita que seja necessário desenvolver um *parser* para objetos remotos, e torna muito mais simples a análise de tais entidades.

Dado um objeto remoto, um *stub* capaz de representar o mesmo possui uma versão de cada método daquele objeto que foi definido em uma interface do tipo `Remote`. A tarefa de `rmec` é, portanto, descobrir todos os métodos remotos que integram um objeto, e fornecer uma implementação para eles no corpo de um *stub*. A implementação do *skeleton*, por outro lado, deve ser capaz de decodificar os argumentos de uma chamada remota e repassá-los para o método adequado no corpo do objeto remoto. Com esse propósito, `rmec` atribui a cada método remoto um código diferente. Mensagens contendo operações remotas são acrescidas com o código da

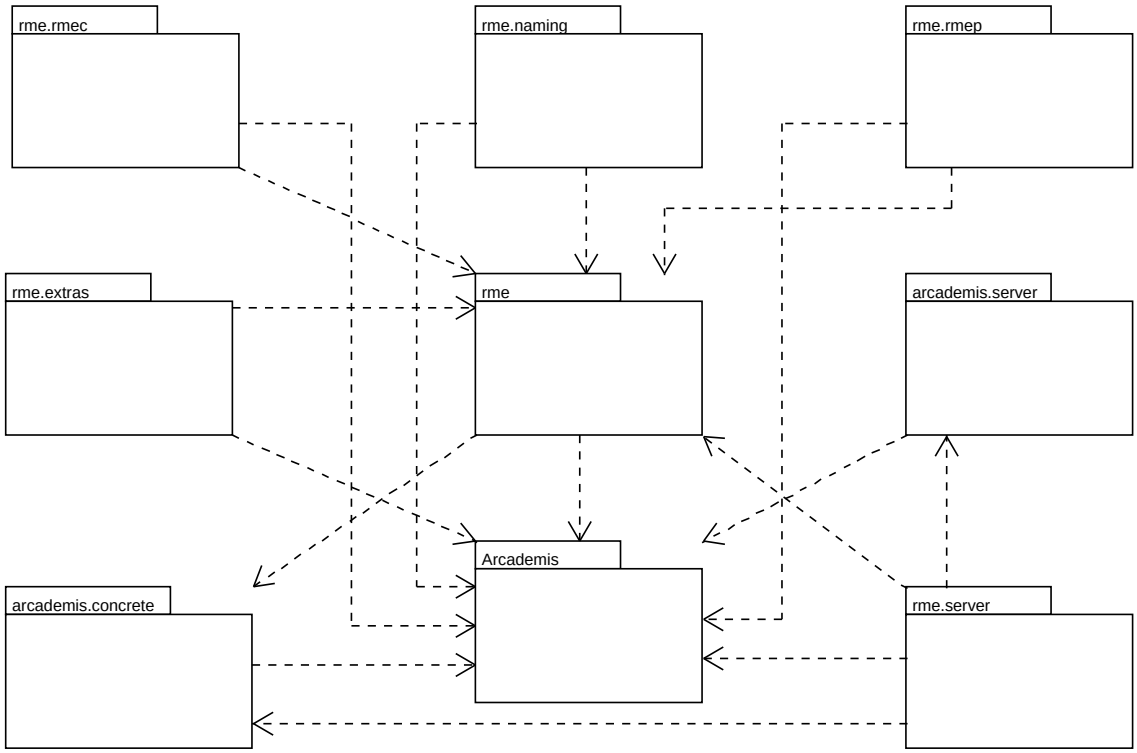

Figura 4.13: Bibliotecas que constituem Arcademis e RME.

operação que representam, de maneira que cada requisição remota possa ser tratada de forma adequada pelo *skeleton*.

4.5.1 Geração de Código para *Stubs*

O *stub*, em RME, possui duas funções principais. A primeira delas é realizar a serialização dos parâmetros de uma invocação remota, e a segunda é permitir a comunicação entre uma aplicação cliente e a implementação de um objeto remoto. Para a serialização, o *stub* utiliza os métodos da classe `RmeStream`, apresentada na Seção 4.3.5. A Figura 4.15 mostra um exemplo de método gerado automaticamente. A implementação a partir da qual tal código foi gerado pode ser vista na Figura 4.14. Conforme pode ser visto nas linhas 5 e 6 da Figura 4.15, cada um dos argumentos de um método remoto são copiados seqüencialmente para um `RmeStream`. O processo inverso, de leitura de um valor de retorno a partir da cadeia de *bytes* recebida também é realizado pelo *stub*. Na Figura 4.15 tal ação acontece na linha 14.

A segunda função do *stub* é desempenhada por um objeto do tipo `Invoker`, que se encarrega de enviar os argumentos serializados para o servidor de chamadas remotas e de repassar para a aplicação cliente o resultado obtido da invocação. A efetiva realização de uma invocação remota acontece no método `invoke`, que na Figura 4.15 está sendo acionado na linha 9. Tal método, implementado em `RmeStub` se encarrega de criar um invocador adequado para a estratégia que

```
// Original method
public int sum(int a, int b) {
  return a + b;
}
```

Figura 4.14: Implementação da operação de soma.

```
// generated method
1: public int sum(int param0, int param1) throws arcademis.ArcademisException {
2:   int resp = 0;
3:   try {
4:     RmeStream args = (RmeStream)RmeOrbAccessor.getStream();
5:     args.write(param0);
6:     args.write(param1);
7:     int op = 3;
8:     // the constant characterizes the invocation strategy
9:     RmeStream future = invoke(args, op, RmeConstants.TWO_WAY_INVOKER);
10:    if(future.isException()) {
11:      Exception e = (Exception)future.readObject();
12:      throw e;
13:    }
14:    resp = (int)future.readInt();
15:  } catch (arcademis.ArcademisException e) {
16:    throw e;
17:  } catch (Exception e) {
18:    throw new arcademis.UnspecifiedException(e.toString());
19:  }
20:  return resp;
21:}
```

Figura 4.15: Método gerado a partir do código visto na Figura 4.14.

lhe foi fornecida. *stubs* podem utilizar diferentes estratégias de invocação. Por exemplo, alguns métodos do *stub* podem empregar invocadores que exigem resposta do servidor, enquanto outros métodos podem usar invocadores que realizam chamadas do tipo *one way*, que não pressupõem valor de retorno.

Cuidado especial foi tomado para a geração de código referente ao tratamento de exceções nos métodos do *stub*. Exceções disparadas no corpo de um método remoto devem ser levantadas pelo método correspondente implementado pelo *stub*. Todos os valores de retorno recebidos pelo *stub* são testados com o intuito de verificar se encerram uma exceção. Um teste deste tipo pode ser visto na linha 10 da Figura 4.15. Assim, para cada tipo diferente de exceção que pode ser levantada pelo método remoto, um bloco do tipo *catch* deve ser gerado na versão do método implementada no *stub*. Tais blocos não podem ser gerados segundo qualquer ordem, pois os blocos que tratam exceções mais específicas devem preceder aqueles que tratam exceções mais gerais, ou o programa final estará sintaticamente incorreto. Por exemplo, constitui erro sintático em Java preceder um bloco que trate subclasses da classe `Exception` por um bloco que trate a própria classe `Exception`. A Figura 4.17 contém a implementação gerada a partir do método mostrado na Figura 4.16, que pode disparar exceções do tipo `NullObjectException`.

```
// Original method
public MarshalableObject getObject(MarshalableObject x)
throws ArcademisException, NullObjectException {
    if(x == null)
        throw new NullObjectException("null object");
    System.out.println(x.toString());
    MarshalableObject y = new MarshalableObject
    (true, 'z', (byte)1, (short)2, (int)4, (long)8, "God by!");
    return y;
}
```

Figura 4.16: Implementação de método remoto.

Observe que um bloco `catch` exclusivo para aquele tipo de exceção aparece nas linhas 18 e 19 da Figura 4.17.

O gerador de *stubs* é um programa implementado em Java, e sua classe principal encontra-se em um arquivo denominado `RmeC.java`. Assim, para invocar o gerador, basta utilizar uma sintaxe como: `java rme.rmec.RmeC  className`, onde `className` é o nome de alguma classe que implementa uma ou mais interfaces do tipo `Remote`. Poder-se-ia perguntar então, por que *stubs* são gerados a partir de classes, e não a partir de interfaces remotas. Existem três respostas complementares para tal fato. A primeira delas diz respeito exatamente à possibilidade de um objeto remoto poder implementar diversas interfaces remotas. Uma vez que um *stub* representa o objeto remoto, e não alguma de suas interfaces, ele deve refletir todo o conjunto de métodos que são definidos por aquele objeto.

Em segundo lugar, cada método do *stub* deve tratar exatamente o conjunto de exceções disparados pelo método remoto implementado. Tal método, embora tenha sido definido em uma interface, não precisa disparar todas as exceções declaradas na definição original. Por exemplo, a interface remota pode conter métodos que disparam exceções do tipo A e B, além de `ArcademisException`. A implementação destes métodos, contudo, não precisa necessariamente disparar qualquer exceção. Uma vez que blocos do tipo `catch` devem ser gerados somente no corpo de métodos que disparam exceções, é necessário conhecer a declaração da implementação do método remoto, e não sua definição em uma interface.

Finalmente, durante a execução de uma aplicação distribuída baseada em RME, há momentos em que é necessário instanciar *stubs* dinamicamente. Tanto *stubs* quanto *skeletons* somente podem ser gerados pela implementação do objeto remoto que eles representam, pois precisam conter o mesmo identificador presente naquele componente. Devido a esse fato, o nome das classes que permitem instanciar *stubs* e *skeletons* é sempre construído da mesma forma: ao nome da implementação do objeto remoto concatena-se as cadeias `_stub` ou `_skeleton`, para a qualificação de *stubs* e *skeletons*, respectivamente. Assim, é necessário que durante a geração de código para *stubs* e *skeletons*, o nome da classe cujas instâncias são responsáveis pela criação de tais entidades seja conhecido.

```
// generated method
1: public MarshalableObject getObject(MarshalableObject param0)
2: throws arcademis.ArcademisException, NullObjectException {
3:
4:   MarshalableObject resp = null;
5:   try {
6:     RmeStream args = (RmeStream)RmeOrbAccessor.getStream();
7:     args.write(param0);
8:     int op = 0;
9:     // the constant characterizes the invocation strategy
10:    RmeStream future = invoke(args, op, RmeConstants.TWO_WAY_INVOKER);
11:    if(future.isException()) {
12:      Exception e = (Exception)future.readObject();
13:      throw e;
14:    }
15:    resp = (MarshalableObject)future.readObject();
16:  } catch (arcademis.ArcademisException e) {
17:    throw e;
18:  } catch (NullObjectException e) {
19:    throw e;
20:  } catch (Exception e) {
21:    throw new arcademis.UnspecifiedException(e.toString());
22:  }
23:  return resp;
24:}
```

Figura 4.17: Método gerado a partir do código mostrado na Figura 4.16.

4.5.2 Geração de Código para *Skeletons*.

Conceitualmente, o *skeleton* é uma entidade de ligação, que permite o contato entre a implementação do objeto remoto e a camada de transporte, de onde são recebidas as chamadas remotas. Um *skeleton* possui somente um método, que é chamado `dispatch`. A função de tal método é decodificar a mensagem que representa uma chamada remota de modo a verificar sua procedência, extrair os parâmetros que a constituem e repassá-los à implementação do objeto remoto de modo que possam ser processados. Além disso, no corpo do método `dispatch` o valor de retorno originado pela invocação remota deve ser serializado antes de ser enviado de volta para o cliente.

Na Figura 4.18 é mostrada a implementação de um objeto remoto, e na Figura 4.19 é apresentado o código do *skeleton* que é gerado para aquele objeto. Analisando-se a Figura 4.19, nota-se que um *skeleton* em essência comporta-se como um demultiplexador de chamadas remotas. Todo *stub* é gerado de forma a atribuir para cada método remoto um código distinto. Esse valor é inserido nas mensagens que representam chamadas remotas, e é utilizado pela implementação do *skeleton* para definir qual operação do objeto remoto deve ser acionada.

Em relação à serialização de objetos, o *skeleton* desempenha papel oposto ao *stub*: ele extrai da cadeia de *bytes* recebida os argumentos de uma chamada remota e serializa o valor de retorno originado pelo processamento daquela operação, quando este existe. Toda chamada remota, em RME, origina uma resposta. Caso o método invocado não possua valor de retorno, a mensagem

```
public class RemoteObject extends RmeRemoteObject implements RemoteInterface {

  public void m1(int i) {
    System.out.println(i);
  }

  public int m2() {
    return 1;
  }
}
```

Figura 4.18: Exemplo de implementação de um objeto remoto.

```
1: public class RemoteObject_Skeleton extends arcademis.server.Skeleton {
2:   public Stream dispatch(RemoteCall r) throws Exception {
3:     RmeRemoteCall remoteCall = (RmeRemoteCall)r;
4:     RmeStream returnStr = (RmeStream)OrbAccessor.getStream();
5:     Stream args = remoteCall.getArguments();
6:     switch (remoteCall.getOperationCode()) {
7:       case 0: {
8:         int param0 = (int)args.readInt();
9:         Marshalable retValue = null;
10:        ((RemoteObject)super.remoteObject).m1(param0);
11:        returnStr.write(retValue);
12:      }
13:      break;
14:      case 1: {
15:        int retValue = ((RemoteObject)super.remoteObject).m2();
16:        returnStr.write(retValue);
17:      }
28:      break;
29:      default: {
30:        throw new ArcademisException("Invalid operation in remote method request");
31:    }
32:  }
33:  return returnStr;
34:}
```

Figura 4.19: Código do *skeleton* gerado para o objeto remoto mostrado na Figura 4.18.

de resposta contem somente um cabeçalho, do contrário contem aquele valor serializado.

4.6 Exemplo de Aplicação: Catálogo Telefônico

A fim de melhor ilustrar a utilização de RME, é mostrada nesta seção uma aplicação simples, desenvolvida para a plataforma J2ME, que utiliza invocação remota de métodos para efetuar consultas a um catálogo telefônico. O objetivo desta aplicação é permitir que usuários de celulares possam efetuar consultas a um banco de dados em que nomes de pessoas estão associados a números de telefone. Tal banco de dados foi implementado como um objeto remoto, e as consultas sobre ele são realizadas via invocação remota de métodos.

Todo objeto remoto, em RME, deve implementar a interface `Remote`. Também em Java RMI objetos remotos devem implementar uma interface que possui este nome. Esta interface não define nenhum método; sua principal utilidade é tornar transparente para o usuário a utilização

```
import rme.*;
import arcademis.*;

public interface PhoneCatalogue extends Remote {
    public PhoneAddress getPhoneAddress(String name)
           throws ArcademisException;
}
```

Figura 4.20: Interface desenvolvida para um catálogo telefônico eletrônico.

```
import arcademis.*;
public class PhoneAddress implements Marshalable {
  private String name = null;
  private String phoneNumber = null;
  private String address = null;
  public PhoneAddress(String name, String phoneNumber, String address){
    this.name = name;
    this.phoneNumber = phoneNumber;
    this.address = address;
  }
  public String getName() {...}
  public String getPhoneNumber() {...}
  public String getAddress() {...}
  public boolean equals(Object o) {...}
  public String toString() {...}
  public void marshal(Stream b) throws MarshalException {
    b.write(phoneNumber);
    b.write(name);
    b.write(address);
  }
  public void unmarshal(Stream b) throws MarshalException {
    this.phoneNumber = (String)b.readObject();
    this.name = (String)b.readObject();
    this.address = (String)b.readObject();
  }
}
```

Figura 4.21: Classe que representa entradas em um catálogo telefônico eletrônico.

de *stubs* em vez da real implementação dos objetos cujos serviços deseja-se invocar. O serviço de nomes devolve objetos do tipo `Remote` como resultado de consultas. A classe `arcademis.Stub`, ancestral de todos os *stubs* em RME também implementa a interface `Remote`. Assim, não existem problemas, em relação ao sistema de tipos de Java, para atribuir a *stubs* o resultado de buscas por objetos remotos. A interface definida para a aplicação deste exemplo pode ser vista na Figura 4.20. Nessa interface, somente um método é definido: `getPhoneAddress`. Tal operação retorna, para cada consulta efetuada com sucesso à lista telefônica, um objeto do tipo `PhoneAddress`, que contém os dados referentes à pessoa pesquisada.

A implementação da classe `PhoneAddress` pode ser vista na Figura 4.21. Uma vez que se tenciona transmitir objetos deste tipo como respostas de invocações remotas, a classe `PhoneAddress` precisa implementar a interface `Marshalable`, e, em conseqüência, ela deve prover um corpo para os métodos `marshal` e `unmarshal`. Objetos do tipo `PhoneAddress` possuem apenas três atributos. O processo de serialização consiste em copiar tais valores para uma cadeia de *bytes*. O processo inverso, executado pelo método `unmarshal`, consiste em preencher os valores dos atributos com informações lidas de uma cadeia de *bytes*.

```
import java.io.*;
import java.util.*;
public class PhoneBook extends rme.server.RmeRemoteObject
implements PhoneCatalogue {
  Hashtable h = null;
  public PhoneBook() {
    h = new Hashtable();
  }
  public PhoneAddress getPhoneAddress(String name) {
    PhoneAddress addr = (PhoneAddress)h.get(name);
    return addr;
  }
  private void insert(String name, String number, String address) {
    PhoneAddress addr = new PhoneAddress(name, number, address);
    h.put(name, addr);
  }
  public void readFile(String fileName) throws IOException {
    // fills the hashtable with information from fileName.
  }
}
```

Figura 4.22: Implementação de um catálogo telefônico eletrônico.

A implementação do catálogo telefônico é simples: trata-se de uma classe que encapsula uma tabela em que estão associados a cada nome de pessoa um número telefônico. Tal implementação é mostrada na Figura 4.22. Observe que esta classe destina-se a ser utilizada remotamente, porém ela não possui qualquer código extra a lhe garantir contato com as primitivas de comunicação do sistema operacional. A não ser por estender `RmeRemoteObject`, esta classe em nada difere de uma classe que somente é utilizada localmente. Toda a implementação necessária para que invocações remotas possam ser recebidas, processadas e respondidas encontra-se no corpo da classe `RmeRemoteObject`. Apenas o método `readFile` não foi integralmente mostrado na Figura 4.22. Tal método permite preencher a tabela com pares formados por um nome de pessoa e um objeto do tipo `PhoneAddress`, cujo conteúdo foi lido a partir de um arquivo.

É necessário implementar um servidor para o catálogo telefônico. Tal servidor terá por função instanciar um objeto do tipo `PhoneBook` e preencher tal objeto com tuplas formadas por nomes, números telefônicos e endereços residenciais. Além disto, cabe ao servidor inicializar as entidades responsáveis pelo recebimento de chamadas remotas. A implementação de tal elemento pode ser vista na Figura 4.23. A configuração do ORB tem lugar nas linhas 9 e 10 dessa figura. No programa apresentado, está sendo utilizada a configuração padrão fornecida por RME, contudo, outras configurações são perfeitamente possíveis. Com tal propósito, poder-se-ia implementar outra classe de configuração, ou poder-se-ia estender a classe `RmeConfigurator`, ou ainda poder-se-ia efetuar modificações no código desta classe. A constatação de que, para ser modificado todo o comportamento da aplicação distribuída, basta alterar uma linha de código (a linha 9 da Figura 4.23) torna nítido o caracter flexível e reconfigurável de Arcademis e da plataforma RME.

Para completar o exemplo apresentado nesta seção, é necessário que seja mostrada a implementação de uma aplicação cliente, isto é, um programa que utiliza os serviços fornecidos

```
1: public class Server {
2:   public static void main(String args[]) {
3:     if(args.length != 2) {
4:       System.err.println("Sintaxe: java PhoneBook objName fileName");
5:       System.exit(1);
6:     }
7:     else {
8:       try {
9:         rme.RmeConfigurator c = new rme.RmeConfigurator();
10:        c.configure();
11:        PhoneBook o = new PhoneBook();
12:        rme.naming.RmeNaming.bind(args[0], o);
13:        o.activate();
14:        o.readFile(args[1]);
15:      } catch (arcademis.ArcademisException e) {
16:        e.printStackTrace();
17:      } catch (arcademis.concrete.MalformedURLException e) {
18:        e.printStackTrace();
19:      } catch (rme.naming.AlreadyBoundException e) {
20:        e.printStackTrace();
21:      } catch (java.io.IOException e) {
22:        e.printStackTrace();
23:      }
24:    }
25:  }
26:}
```

Figura 4.23: Implementação de um servidor de métodos remotos.

pela classe `PhoneBook`, vista na Figura 4.22. Tal classe foi implementada sobre a plataforma J2ME, e pode ser utilizada sobre um emulador de celular, ou sobre um dispositivo real, caso este suporte CLDC. Dado que tal implementação é constituída por um número relativamente grande de linhas de código, ela é apresentada em partes. A estrutura geral do código pode ser vista na Figura 4.24. O método construtor e o método responsável pelo tratamento de eventos encontram-se implementados, respectivamente, nas Figuras 4.25 e 4.26.

A aplicação cliente consiste basicamente de um formulário no qual podem ser informados o caminho até um objeto e um nome para consulta. Na Figura 4.24 foram mostrados apenas os comandos necessários para construir o formulário, que é formado por quatro campos: *url*, *name*, *phone* e *addr*. A primeira destas entradas especifica a localização de um objeto remoto, a qual é definida por um endereço IP seguido por um nome de objeto. O segundo campo permite que o usuário forneça nomes para consulta e os outros dois campos são utilizados pela aplicação para fornecer os resultados obtidos em uma consulta remota.

O método construtor pode ser visto na Figura 4.25. O trecho de código compreendido entre as linhas 8 e 10 diz respeito à configuração do *middleware* e à busca por um objeto distribuído identificado pelo nome *obj* e localizado em um *host* cujo endereço IP é `algol.dcc.ufmg.br/obj`. Os demais comandos concernem a inicialização de outros componentes da aplicação, como a tela gráfica e botões de comando.

```
1: import java.io.*;
2: import javax.microedition.io.*;
3: import javax.microedition.midlet.*;
4: import javax.microedition.lcdui.*;
5: import rme.*;
6: import rme.naming.*;
7: import arcademis.*;
8:
9: public class PhoneBookClient extends MIDlet implements CommandListener {
10:  private Display display = null;
11:  private PhoneCatalogue phoneBook = null;
12:  private Command exitCmd = null;
13:  private Command getInf = null;
14:  private Form sendScr = null;
15:  private TextField url = null;
16:  private TextField name = null;
17:  private TextField phone = null;
18:  private TextField addr = null;
19:
20:  public PhoneBookClient() { ... }
21:
22:  private Screen showSendScreen() {
23:    if(sendScr == null) {
24:      sendScr = new Form("Send Message");
25:      sendScr.addCommand(exitCmd);
26:      sendScr.addCommand(getInf);
27:      sendScr.setCommandListener(this);
28:      url = new TextField("URL: ", "algol.dcc.ufmg.br/obj", 60, TextField.URL);
29:      sendScr.append(url);
30:      name = new TextField("Name: ", "", 40, TextField.ANY);
31:      sendScr.append(name);
32:      phone = new TextField("Phone: ", "0", 20, TextField.ANY);
33:      sendScr.append(phone);
34:      addr = new TextField("Address: ", "0", 60, TextField.ANY);
35:      sendScr.append(addr);
36:    }
37:    display.setCurrent(sendScr);
38:    return sendScr;
39:  }
40:  public void startApp() {
41:    showSendScreen();
42:  }
43:  public void pauseApp() {
44:  }
45:  public void destroyApp(boolean unconditional) {
46:  }
47:  public void commandAction(Command c, Displayable s) { ... }
48:}
```

Figura 4.24: Aplicação que realiza consultas telefônicas em J2ME.

```
1: public PhoneBookClient() {
2:   display = Display.getDisplay(this);
3:
4:   exitCmd = new Command("Exit", Command.EXIT, 1);
5:   getInf = new Command("getInf", Command.OK, 2);
6:
7:   try {
8:     RmeConfigurator mc = new RmeConfigurator();
9:     mc.configure();
10:    phoneBook = (PhoneCatalogue)RmeNaming.lookup("algol.dcc.ufmg.br/obj");
11:  }
12:  catch (Exception e) {
13:    e.printStackTrace();
14:  }
15:}
```

Figura 4.25: Método construtor para a aplicação cliente mostrada na Figura 4.24.

A Figura 4.26 apresenta a implementação do método `commandAction`, o qual é responsável pelo tratamento de eventos que são disparados pelo usuário enquanto este interage com o dispositivo móvel. Dois tipos de eventos são tratados pelo método da Figura 4.26: são eles o encerramento da aplicação e a ativação de um comando nomeado `getInf`. CLDC permite que botões de comando sejam definidos e adicionados à aplicação a fim de que o usuário tenha como solicitar a realização de algumas ações. Quando o evento de encerramento é acionado, os recursos alocados pela aplicação são liberados de volta para o dispositivo móvel. Quando o comando `getInf` é acionado, uma consulta é realizada sobre o objeto remoto que implementa o catálogo telefônico. Tal consulta fornece o nome que deve ser informado na caixa de texto *name* e recebe como resposta um objeto do tipo `Address`, que contém as informações associadas àquele nome na lista telefônica.

4.7 Análise de Desempenho de RME

Esta seção apresenta alguns experimentos realizados com o intuito de avaliar o desempenho das versões de RME desenvolvidas para J2ME e J2SE. Todos os testes utilizaram a interface mostrada na Figura 4.27. Esta interface define quatorze diferentes métodos, os quais podem ser divididos em três grupos. Destes, o primeiro contém métodos que não apresentam argumentos e que possuem um valor de retorno diferente de `void`. O nome de tais métodos é formado pelo prefixo `get` seguido do nome do tipo de retorno. Fazem parte desse grupo rotinas como `getByte` e `getShort`. O segundo grupo é composto por métodos que possuem como único argumento um arranjo de tipos primitivos ou de *strings*. Pertencem a este grupo os métodos cujos nomes são iniciados pelo prefixo `pass`, exceto `passArgs`, que é o único integrante do último grupo.

Procurou-se fornecer para os métodos da Figura 4.27 implementações simples, cujo tempo de processamento não viesse a interferir no desempenho das rotinas utilizadas por RME para permitir a comunicação entre diferentes máquinas virtuais. Assim, a implementação dos métodos

```
1: public void commandAction(Command c, Displayable s) {
2:   try {
3:     if(s == sendScr) {
4:       if (c == exitCmd) {
5:         destroyApp(false);
6:         notifyDestroyed();
7:       }
8:       else if (c == getInf) {
9:         PhoneAddress a = phoneBook.getPhoneAddress(name.getString());
10:        if(a != null) {
11:          phone.setString(a.getPhoneNumber());
12:          addr.setString(a.getAddress());
13:        } else {
14:          name.setString("Information not found.");
15:        }
16:      }
17:    }
18:  } catch (Exception e) {
19:    e.printStackTrace();
20:  }
21:}
```

Figura 4.26: Tratamento de eventos para a aplicação vista na Figura 4.24.

do tipo `get`, como por exemplo `getByte()`, contém somente um comando de retorno. Já a implementação dos métodos que recebem arranjos como parâmetros de entrada converte cada uma das células de seus argumentos para objetos do tipo `String`, que são concatenados em um outro `String`, o qual é retornado para a aplicação cliente. Por fim, a implementação de `passArgs` possui corpo nulo.

4.7.1 Desempenho de RME para J2ME

Para medir o desempenho da versão de RME para J2ME, a aplicação cliente foi executada sobre um emulador de telefone celular cuja máquina virtual K (KVM), executa 100 *bytecodes* por milisegundo. Tal emulador foi executado em um computador do tipo *Pentium* 4 de 2.0GHz e 512MB de memória disponível. Uma máquina de igual capacidade foi utilizada para a execução do servidor de serviços, que implementa a interface mostrada na Figura 4.27. As aplicações cliente e servidora foram executadas em máquinas diferentes, conectadas por uma rede *Ethernet* de 10Mb/s.

A fim de fornecer um limite superior para o maior número de requisições que podem ser tratadas no cenário descrito anteriormente, foi implementada uma aplicação que troca seqüências de *bytes* de mesmo tamanho que aquelas trocadas no teste realizado com RME, sem, contudo, realizar qualquer processamento sobre tais cadeias. Esta aplicação faz uso da mesma implementação de canal empregada em RME: uma conexão baseada em `sockets TCP/IP` que utiliza um *buffer* de dados para permitir o agrupamento de *bytes* antes da transmissão dos mesmos. Tanto a aplicação baseada em `sockets` quanto aquela baseada em RME são constituídas por

```
import arcademis.*;
public interface MethodSet extends Remote {
  public byte getByte() throws ArcademisException;
  public short getShort() throws ArcademisException;
  public char getChar() throws ArcademisException;
  public int getInt() throws ArcademisException;
  public long getLong() throws ArcademisException;
  public String getString() throws ArcademisException;
  public String[] getStrs() throws ArcademisException;

  public void passArgs(byte b, short sh, char c, int i,
  long l, String st, String[] sts) throws ArcademisException;

  public String passBytes (byte[] b) throws ArcademisException;
  public String passShorts (short[] s) throws ArcademisException;
  public String passChars (char[] c) throws ArcademisException;
  public String passInts (int[] i) throws ArcademisException;
  public String passLongs (long[] l) throws ArcademisException;
  public String passStrs (String[] s) throws ArcademisException;
}
```

Figura 4.27: Interface remota utilizada nos testes de desempenho.

um cliente implementado em J2ME e por um servidor implementado em J2SE. Os resultados mostrados na Tabela 4.5 representam o número de requisições tratadas por segundo por RME e pela implementação baseada em `sockets`. Para cada um dos métodos mostrados na tabela foram realizadas 10 seqüências de 50 invocações. A tabela contém o resultado médio apurado.

O número relativamente pequeno de requisições tratadas deve-se à pequena capacidade de processamento do emulador utilizado. Dado que o emulador permite a execução de somente 100 *bytecodes* por milisegundo, o tempo necessário para inicializar os componentes necessários e serializar objetos passa a ser relevante em relação ao tempo gasto para a transmissão de dados. Métodos que possuem objetos como argumentos ou valores de retorno, conforme pode ser apurado pela análise da Tabela 4.5, apresentam pior desempenho que os métodos que apenas causam a serialização de tipos primitivos.

Nota-se também que os tempos de execução da aplicação baseada em `sockets` permanecem estáveis. Uma vez que tal aplicação apenas envia e recebe *bytes*, sem realizar qualquer processamento sobre eles, somente o tamanho das mensagens trocadas irá influenciar seu tempo de execução. Por outro lado, os *bytes* que compõem uma mensagem são enfileirados, a fim de serem transmitidos todos de uma vez, de modo que somente um acesso a camada de transporte é feito pela aplicação cliente para cada simulação de invocação remota. O enfileiramento de *bytes* tende a estabilizar o tempo de transmissão de informações. Este somente irá apresentar grande variação se o tamanho das mensagens enviadas variar substancialmente.

4.7.2 Desempenho de RME para J2SE

A fim de testar a versão de RME para a plataforma J2SE foram utilizadas as mesmas máquinas usadas no experimento anterior: dois computadores do tipo Pentium 4 conectados por uma rede Ethernet de 10Mb/s e dotados de processadores de 2.0GHz e 512MB de memória. Também neste experimento a base de testes empregada foi a mesma: o conjunto de métodos

método	Tamanho da Requisição	Tamanho do Retorno	**req/s RME**	**req/s socket**	**RME/socket**
getShort	200	70	5.08	5.11	0.99
getChar	200	70	5.05	5.12	0.98
getInt	200	72	5.06	5.11	0.99
getLong	200	76	5.02	5.07	0.99
getString	200	81	4.75	5.01	0.95
getStrs	200	201	2.96	5.00	0.59
passArgs	351	69	2.57	5.02	0.52
passBytes	213	82	4.48	5.05	0.90
passShorts	223	82	4.26	5.02	0.85
passChars	223	90	4.11	5.04	0.82
passInts	243	82	3.93	5.08	0.78
passLongs	282	82	3.03	5.04	0.61
passStrs	323	170	2.12	4.97	0.43

Tabela 4.5: Número de requisições/s efetuadas por cliente RME/J2ME.

mostrado na Figura 4.27. Neste experimento, contudo, tanto a aplicação servidora quanto a aplicação cliente foram implementadas em J2SE.

No experimento apresentado na Seção 4.7.1 foi necessário desenvolver uma aplicação que servisse como parâmetro de eficiência, pois não há outra implementação de um serviço de invocação remota de métodos para a configuração CLDC que pudesse ser comparado com RME. Para a plataforma J2SE, contudo, existem diversas implementações de chamada remota de métodos, sendo Java RMI a mais conhecida. Tal plataforma foi, assim, adotada para efeitos de comparação com a implementação de RME. A mesma implementação para a interface apresentada na Figura 4.27 foi fornecida para ambas as plataformas.

Foram realizados testes locais, isto é, em que tanto o servidor quanto a aplicação cliente encontram-se localizados no mesmo computador, e testes em rede. Os resultados dos testes locais e em rede podem ser vistos nas Tabelas 4.6 e 4.7, respectivamente. Os tamanhos de requisições e valores de retorno são os mesmos que os mostrados na Tabela 4.5. Para cada um dos métodos apresentados nas tabelas, é mostrado o resultado médio apurado a partir da execução de 10 séries de 10000 requisições remotas. Tal resultado representa o número de invocações por segundo realizadas sobre cada método.

Os experimentos cujos resultados são apresentados na Tabela 4.6 permitem comparar o desempenho das técnicas de serialização de objetos utilizadas em Java RMI e em RME. Métodos que simplesmente retornam valores primitivos comportam-se melhor em Java RMI. Por outro lado, métodos que possuem como argumentos de entrada ou valores de retorno algum tipo de dado composto, isto é, que é instância de alguma classe, tendem a ser executados mais rapidamente em RME. Nota-se, por exemplo, que todos os métodos que retornam objetos do tipo `String` possuem melhores taxas de execução em RME. A técnica de serialização utilizada

método	**req/s RME**	**req/s RMI**	RME/RMI
getShort	4080.80	5652.11	0.72
getChar	4177.98	5765.35	0.72
getInt	4133.94	5884.95	0.70
getLong	4142.50	5885.82	0.70
getString	4058.99	5099.44	0.79
getStrs	3348.03	2729.07	1.23
passArgs	3226.50	2755.58	1.17
passBytes	3942.18	3144.90	1.25
passShorts	3917.98	3303.60	1.18
passChars	3798.43	3200.00	1.19
passInts	3851.34	3236.25	1.19
passLongs	3671.27	3114.54	1.18
passStrs	2891.57	2431.02	1.19

Tabela 4.6: Comparação entre RME e Java RMI – Testes locais.

método	**req/s RME**	**req/s RMI**	**RME/RMI**
getShort	1814.31	2691.97	0.67
getChar	1831.17	2749.71	0.67
getInt	1818.26	2746.69	0.66
getLong	1802.86	2711.50	0.66
getString	1782.61	2557.71	0.70
getStrs	1405.88	1555.27	0.90
passArgs	1208.71	1512.11	0.80
passBytes	1714.68	2057.82	0.83
passShorts	1625.29	2074.69	0.78
passChars	1504.27	2022.14	0.74
passInts	1466.97	2012.47	0.73
passLongs	1376.37	1859.17	0.74
passStrs	1100.11	1329.17	0.82

Tabela 4.7: Comparação entre RME e Java RMI em rede Ethernet de 10Mb/s.

em RME é mais eficiente porque os métodos para serializar e recuperar o conteúdo de objetos são implementados diretamente pelo desenvolvedor de aplicações. O algoritmo de serialização adotado por Java RMI é muito mais geral, uma vez que não exige do desenvolvedor nenhum esforço adicional, porém a utilização de reflexividade o torna mais lento.

Os experimentos realizados em rede, e cujos resultados são mostrados na Tabela 4.7, fornecem uma idéia acerca do desempenho da implementação da camada de comunicação de RME. Tal implementação não é tão eficiente quanto aquela utilizada em Java RMI, porém o seu objetivo principal não é superar o sistema da *Sun*: é demonstrar que a flexibilidade não compromete de modo substancial o seu desempenho.

Um dos motivos que leva a invocação remota de métodos em RME a ser mais lenta que operações análogas em Java RMI são as indireções adicionadas pelo uso de fábricas de objetos, decoradores e processadores de serviço. Tais facilidades, ausentes em Java RMI, tornam RME uma plataforma muito mais flexível, porém são responsáveis por maiores atrasos durante a invocação remota de métodos. Em RME, a flexibilidade cobra seu preço, tanto em termos de eficiência, conforme pode ser constatado pela análise das Tabelas 4.6 e 4.7, quanto em termos do tamanho de suas bibliotecas, conforme tratado na Seção 5.2.2.

4.8 Conclusão

Este capítulo apresentou RME, um serviço de invocação remota de métodos desenvolvido a partir de Arcademis para a configuração CLDC da plataforma J2ME. Em particular, este capítulo procurou discutir as estratégias de implementação adotadas no projeto deste sistema, ou seja, como os componentes abstratos de Arcademis foram implementados e quais componentes concretos do arcabouço puderam ser reaproveitados. A fim de demonstrar a viabilidade de RME, foram apresentados testes em que o desempenho deste sistema é analisado nas plataformas J2ME e J2SE. Neste último ambiente, RME é comparado com Java RMI, e, embora o sistema obtido de Arcademis não seja tão eficiente quanto a implementação da *Sun*, os testes mostraram que a maior flexibilidade de RME não compromete substancialmente o seu desempenho. A implementação de RME constitui por si só uma contribuição desta pesquisa na medida em que não existem outros serviços de invocação remota de métodos para a configuração CLDC de J2ME. Além disto, RME permite evidenciar a flexibilidade de Arcademis, uma vez que este arcabouço foi utilizado em um ambiente (J2ME/CLDC) no qual a implementação tradicional de Java RMI não pode ser empregada.

Capítulo 5

Avaliação Final e Conclusões

Neste capítulo, o arcabouço proposto é avaliado segundo três critérios: flexibilidade, facilidade de uso e generalidade. Além disto, são apresentadas comparações entre Arcademis e Quarterware e comparações entre RME, Java RMI e outros sistemas de *middleware*, em particular, as plataformas reconfiguráveis apresentadas no Capítulo 2.

5.1 Avaliação de Arcademis

A fim de avaliar o arcabouço desenvolvido, são utilizados três critérios: flexibilidade, facilidade de uso e generalidade. Embora diferentes, a definição destes parâmetros de avaliação se sobrepõem em alguns pontos. A flexibilidade é um conceito amplo, pois engloba questões relacionadas à reusabilidade dos componentes do arcabouço e também questões relacionadas à generalidade do mesmo, isto é, em quais cenários ele pode ser empregado. A flexibilidade de um arcabouço é tanto maior quanto maiores forem as possibilidades dele ser utilizado na instanciação de sistemas que atendem a diferentes requisitos. A facilidade de uso é uma medida de quão simples é derivar plataformas de *middleware* a partir do arcabouço ou a partir de alguma de suas instâncias.

5.1.1 Flexibilidade.

Arcademis é, essencialmente, um arcabouço voltado para o desenvolvimento de plataformas de *middleware* orientadas por objetos, e, embora alguns de seus componentes possam ser utilizados para implementar sistemas baseados em outros paradigmas, como troca de mensagens, por exemplo, este não é o objetivo central do arcabouço. Assim sendo, Arcademis apresenta maior flexibilidade quando necessário instanciar sistemas de *middleware* que sejam baseados no modelo introduzido pelos objetos de rede de Modula-3, que foi apresentado na Seção 2.1.

Sistemas de *middleware* derivados de Arcademis são muito mais flexíveis, por exemplo, que a plataforma Java RMI da *Sun*. Conforme discutido na Seção 2.2.3, RMI apresenta algumas opções de reconfiguração, principalmente relacionadas à implementação do canal de comunicação

utilizado, porém estas são muito limitadas. Assim, nesta plataforma é possível aplicar um algoritmo de criptografia ou de compactação às mensagens que estão sendo transmitidas por *stubs* e *skeletons*, porém não é possível fazer com que o sistema utilize outra técnica de serialização que não aquela baseada em reflexividade. Tampouco é possível modificar de modo simples alguns aspectos da arquitetura de Java RMI, como, por exemplo, a política de *threads* adotada pelos servidores, ou a semântica de invocação remota.

Java RMI, contudo, não é um arcabouço de desenvolvimento de *software*, mas um *middleware* pronto. Ainda assim, modificar plataformas de *middleware* derivadas de Arcademis, como por exemplo RME, é mais fácil que alterar componentes de Java RMI. Tal facilidade deve-se, principalmente, ao projeto de Arcademis, segundo o qual componentes são criados por meio de fábricas de objetos, e não diretamente. Desta forma, para modificar algum componente específico de um sistema derivado, basta alterar a fábrica responsável pela criação daquele componente. Tal alteração, em geral, pode ser feita sem que seja necessário modificar outras partes do sistema.

Além disso, a arquitetura de Arcademis foi desenvolvida segundo diversos padrões de projeto, muitos dos quais prevêem reconfigurações. O padrão *acceptor-connector*, por exemplo, torna simples a implementação de conexões síncronas ou assíncronas, isto é, que não causam o bloqueio da aplicação. Os quatro processadores de serviço que se interpõem entre *stubs* e *skeletons*, vistos na Seção 3.1.2, por sua vez, permitem que diferentes tipos de semânticas para chamadas remotas sejam utilizadas. Além disso, tais componentes facilitam a alteração da estrutura do servidor de chamadas remotas, permitindo, por exemplo, que sejam definidos servidores baseados em uma única *thread* ou em várias.

Java RMI permite que a implementação de canais de comunicação seja modificada via alterações em um componente denominado `SocketFactory`. Este *middleware* usa uma fábrica de `sockets` padrão, sendo possível especificar outra implementação para a mesma durante a instanciação de um objeto remoto. Para modificar a implementação do canal utilizado em Arcademis, basta alterar a fábrica responsável pela criação deste componente. Além disso, Arcademis utiliza o padrão conhecido por *decorator* para adicionar novas funcionalidades aos canais de comunicação. Diversos decoradores podem ser utilizados em conjunto e em qualquer ordem, o que se mostra uma abordagem bem mais flexível que aquela adotada em Java RMI.

A facilidade com que o protocolo de comunicação pode ser alterado também é uma medida importante acerca da flexibilidade de uma plataforma de *middleware*. Protocolos de comunicação, em Arcademis, são definidos como conjuntos de classes que implementam a interface `Message`. Cada mensagem é responsável pela sua codificação e decodificação em *bytes*. Conforme discutido na Seção 3.1.5, para alterar o protocolo de comunicação adotado, basta criar novas mensagens, ou destruir mensagens antigas e modificar os processadores de serviço que implementam o padrão *request-response* (Seção 3.1.6). Uma vez que todas as definições de mensagens e estados possíveis do protocolo estão definidos nestes componentes, alterá-los não causa

impactos a outras partes da plataforma. Em Java RMI, por outro lado, não é possível alterar JRMP, seu protocolo de comunicação, pois a implementação deste não se encontra nem mesmo disponível junto aos pacotes que acompanham a distribuição padrão de Java.

Finalmente, RME, uma instância de Arcademis, pôde ser utilizado em um ambiente onde o mecanismo de reflexividade não está disponível. Este fato evidencia a flexibilidade do arcabouço, pois ele foi empregado em um cenário no qual o modelo de serialização de Java não pode ser aplicado.

5.1.2 Facilidade de Uso

Quantificar a facilidade de uso de um arcabouço de desenvolvimento de *software* não é uma tarefa que pode ser realizada facilmente, pois envolve critérios muitas vezes subjetivos. Usuários familiarizados com a arquitetura de Arcademis provavelmente poderiam desenvolver plataformas de *middleware* mais rapidamente utilizando o arcabouço em vez de o fazerem sem uma base inicial, pois eles teriam acesso a diversos componentes já implementados e testados e, mais importante, teriam acesso a um projeto pronto, isto é, à maneira como um conjunto relativamente grande de componentes estão relacionados. Por outro lado, a fim de poder ser efetivamente utilizado, o arcabouço precisa ser conhecido pelo desenvolvedor de aplicações, e tal aprendizado exige algum tempo.

Outra maneira de utilizar Arcademis é derivar sistemas de *middleware* a partir de alguma de suas instâncias, como RME, por exemplo. Caso as diferenças entre a plataforma original e a desejada não sejam grandes, este processo torna ainda mais eficiente o processo de desenvolvimento de *middleware*. Por outro lado, quando utilizando uma das instâncias de Arcademis como ponto de partida, o desenvolvedor precisa conhecer, além da estrutura do arcabouço, algumas particularidades do *middleware* que está sendo utilizado.

Arcademis possui componentes concretos que podem ser facilmente utilizados na composição de plataformas de *middleware*. Por exemplo, para modificar a arquitetura de RME, de modo que fosse utilizada uma única *thread* para o processamento de chamadas ao invés de várias, não é necessário implementar novas classes. Bastaria, neste caso, modificar a classe `RmeConfigurator` para associar ao `ORB` uma fábrica de processadores de serviço diferente daquela normalmente utilizada, além de uma fábrica de filas e outra de escalonadores, sendo que todos estes componentes já estão implementados.

Embora Arcademis defina uma série de componentes concretos, muitos de seus componentes são utilizados segundo o princípio de arcabouços caixa-branca, introduzidos na Seção 2.5.1. Desta forma, alguns dos componentes de Arcademis são classes abstratas, que o desenvolvedor precisa estender e implementar. Tal fato exige do desenvolvedor familiaridade com o arcabouço, pois este precisa lidar com atributos internos de alguns componentes. É possível, contudo, que plataformas de *middleware* derivadas de Arcademis, como RME, sejam empregadas como exemplo durante a implementação de componentes abstratos.

A utilização de fábricas facilita a reconfiguração de instâncias de RME. Basta alterar a fábrica responsável pela criação de algum componente do sistema para modificar toda a semântica da plataforma de *middleware* que depende do componente alterado. Também a arquitetura do arcabouço, baseada em uma série de padrões de projeto bem conhecidos, é outro fator que contribui para tornar mais simples a modificação de suas instâncias. Conforme discutido na Seção 2.4, alguns padrões de projeto já prevêem alterações a fim de lidar com especificações de requisitos diferentes.

Espera-se que Arcademis seja utilizado na instanciação de outras plataformas de *middleware* diferentes de RME. A partir destes outros sistemas, uma análise mais profunda a respeito da facilidade de uso do arcabouço e do benefício que advém de sua utilização poderá ser desenvolvida. A partir de dois sistemas obtido de Arcademis é possível realizar experimentos como, por exemplo, a mensuração de quanto do arcabouço foi de fato reutilizado em ambas as instâncias, tanto em termos de linhas de código quanto em termos de número de componentes.

5.1.3 Generalidade

No contexto desta dissertação, um arcabouço para o desenvolvimento de *software* é tão mais genérico quanto maiores forem as diferenças possíveis entre suas instâncias. A generalidade de um sistema de *middleware*, por sua vez, de acordo com este mesmo conceito, pode ser medida pela quantidade de cenários diferentes com os quais ele pode lidar. Nesse sentido CORBA apresenta maior generalidade que Java RMI, pois aquela plataforma pode suportar aplicações escritas em diferentes linguagens, ao passo que o sistema da *Sun* está restrito à linguagem Java.

Arcademis foi proposto com o objetivo primordial de facilitar a implementação de sistemas de *middleware* baseados em invocação remota de métodos, porém este arcabouço também pode ser utilizado no desenvolvimento de plataformas baseadas em outros paradigmas, diferentes do orientado por objetos. A fim de ilustrar tal possibilidade, esta seção mostra como Arcademis pode ser usado no desenvolvimento de uma implementação de PeerSpaces [VBBP02]. Este sistema é um *middleware* voltado para a coordenação de aplicações em redes móveis ad-hoc, que baseia-se em uma estrutura de dados denominada espaço de tuplas.

Espaços de tuplas, introduzidos por David Gelernter na especificação da linguagem Linda [Gel85], são estruturas de dados presentes em cada um dos nodos que integram uma rede em PeerSpaces, e onde podem ser disponibilizados serviços e informações. Tuplas são listas ordenadas de campos, cada campo possuindo um tipo bem definido, como `int` ou `java.lang.String`, por exemplo. Diversos nodos podem interagir procurando, lendo, escrevendo e consumindo informações em seus próprios espaços ou em repositórios de seus vizinhos, o que é feito por meio de operações básicas. Tais operações são descritas por um conjunto de quatro primitivas: **in**, **out**, **rd** e **find**. A primeira operação remove e a segunda insere dados no espaço de tuplas. O comando **rd** realiza a leitura de dados e a primitiva **find** permite realizar operações de busca na rede de computadores.

Uma implementação de PeerSpaces baseada em Arcademis pode valer-se do modelo de objetos distribuídos utilizado pelo arcabouço. Cada *host*, isto é, cada entidade que detém o controle de um espaço de tuplas, seria implementado como um objeto remoto, cujos métodos são as quatro primitivas definidas em PeerSpaces. Como cada *stub* possuiria somente quatro operações, o código destes componentes poderia ser otimizado a fim de obter benefício de algumas particularidades de cada primitiva. Por exemplo, segundo a especificação de Arcademis, chamadas remotas possuem identificadores. Estes poderiam ser usados para evitar ciclos durante a propagação de buscas na rede via o comando **find**.

Como é possível notar, embora Arcademis tenha sido projeto a fim de permitir a instanciação de sistemas de *middleware* orientados por objetos, ele pode ser utilizado no desenvolvimento de plataformas que obedecem paradigmas diferentes. Outro elemento que vem contribuir para demonstrar a generalidade deste Arcabouço é o fato de ele poder ser utilizado em J2ME, uma distribuição da linguagem Java diferente de J2SE, a versão em que foi desenvolvido Java RMI. A fim de alcançar tal objetivo, a implementação de Arcademis utiliza somente tipos primitivos, interfaces e classes que são comuns a todas as distribuições de Java. Assim, nenhum dentre os componentes de Arcademis utiliza tipos como *float* ou *double*, uma vez que eles não se encontram disponíveis para Java *Micro Edition*.

Contudo, Arcademis ainda está restrito à linguagem Java, muito embora este arcabouço possa ser utilizado no desenvolvimento de diferentes tipos de *middleware* e nas três diferentes edições desta linguagem de programação: J2ME, J2SE e J2EE. Assim, sistemas de *middleware* derivados de Arcademis são implementados em Java e aplicações distribuídas que são executadas sobre tais plataformas também devem ser implementadas nesta linguagem. Esta última restrição não se aplica a CORBA ou .NET, por exemplo, pois tais plataformas suportam objetos implementados em diferentes linguagens de programação, desde que estas possuam um mapeamento para a linguagem de definição de interfaces fornecida por cada modelo.

Esta restrição de Arcademis, embora uma desvantagem, em termos de generalidade, apresenta alguns pontos positivos. Em primeiro lugar, sendo um sistema voltado exclusivamente para a linguagem Java, Arcademis pode tirar proveito de todos os recursos mais importantes desta linguagem, como a resolução de tipos e o carregamento de classes em tempo de execução, por exemplo. A fim de melhor ilustrar tal vantagem, pode-se considerar a implementação de RMI derivada de Quarterware. Conforme discutido na Seção 2.5.2, nesta plataforma, apenas objetos cujo tipo é conhecido em tempo de compilação podem ser serializados. Esta limitação se deve ao fato de Quarterware ter sido implementado em C++, não havendo nesta linguagem carregadores dinâmicos de classes, os quais permitiriam que um cliente pudesse ter acesso aos *bytecodes* de uma classe localizada em outro *host*.

Outra vantagem que advém do fato de instâncias de Arcademis apenas precisarem lidar com aplicações implementadas em Java é a simplicidade das mesmas, pois não é preciso que tais plataformas de *middleware* interajam com linguagens de definição de interface, como IDL de

CORBA. Tal fato evita a necessidade de serem desenvolvidos tradutores e torna mais simples o acoplamento entre as aplicações distribuídas e o *middleware* subjacente.

5.2 Considerações Finais

5.2.1 Arcademis e Quarterware

Quarterware, apresentado na Seção 2.5.2, assim como Arcademis, é um arcabouço voltado para o desenvolvimento de plataformas de *middleware*. Também como Arcademis, Quarterware é formado por um núcleo de componentes que devem fazer parte de todo *middleware* dele derivado e por um conjunto de componentes opcionais. As diferenças entre os dois sistemas, contudo, são várias. A maior destas diferenças é o fato de Quarterware ter sido implementado em C++ enquanto Arcademis foi implementado em Java. Além disto, estruturalmente Arcademis e Quarterware são sistemas muito diferentes, conforme discutido na Seção 2.5.2.

Quarterware, sendo um sistema mais antigo que Arcademis, apresenta algumas capacidades que este arcabouço ainda não possui, como a possibilidade de utilizar reflexão computacional para realizar alterações dinâmicas em alguns de seus aspectos. Além disso, existem mais exemplos de instâncias de Quarterware que de Arcademis. Há, inclusive, uma instância de Quarterware para Java RMI, embora esta, devido ao fato de C++ não possuir todos os recursos de Java, apresente capacidades bem mais limitadas que a versão original [SSC98, Sin99].

5.2.2 O Tamanho das Bibliotecas de Arcademis/RME

A Tabela 5.1 compara o tamanho das bibliotecas de RME com bibliotecas de outros sistemas de *middleware* reconfiguráveis: UIC CORBA e LegORB. A plataforma UIC foi descrita na Seção 2.3.2 como um exemplo de *middleware* que pode ser reconfigurado dinamicamente. O sistema LegORB [RMKC00], por sua vez, pode ser considerado um precursor de UIC, e também permite reconfigurações dinâmicas e estáticas. Estes dois exemplos de *middleware* foram implementados em C++ e são compatíveis com o sistema CORBA. O tamanho das bibliotecas de Arcademis e RME é o tamanho, em *bytes*, dos arquivos `.jar` que compõem estes pacotes. O tamanho da máquina virtual Java (KVM) utilizada para executar RME não está incluído nos dados mostrados na Tabela 5.1. Contudo, este programa não ocupa mais que 32KB [RTV01]. O tamanho das outras plataformas (UIC e LegORB) é definido pelo tamanho dos arquivos binários que as constituem. Tais arquivos foram obtidos a partir da compilação de programas escritos na linguagem C++.

As bibliotecas que formam RME são compostas por componentes de Arcademis e por um segundo pacote de classes, cujos elementos estendem as classes ou implementam as interfaces do primeiro grupo e que constituem o pacote RME propriamente dito. Os elementos do pacote Arcademis em muitos casos não fornecem qualquer funcionalidade à plataforma de *middleware*, pois são interfaces ou classes abstratas; porém, conforme é discutido na Seção 5.3, tais componentes

Arcademis Cliente	19.2KB
Arcademis/RME Cliente sem fábricas	29.2KB
Arcademis/RME Cliente	37.1KB
Arcademis/RME Servidor	68.2KB
LegORB configurado estaticamente	22.5KB
LegORB configurado dinamicamente	141.5KB
UIC CORBA (PalmOS)	18KB
UIC CORBA (Windows CE(SH3))	29KB
UIC CORBA (Windows 2000)	72KB

Tabela 5.1: Tamanho das bibliotecas dos sistemas LegORB, UIC CORBA e RME.

são responsáveis por dotar RME de grande flexibilidade, no sentido de que este *middleware* pode ser facilmente alterado para atender a diferentes especificações de requisitos.

A flexibilidade de RME possui um custo em termos do tamanho de suas bibliotecas, pois, conforme pode ser observado na Tabela 5.1, o pacote Arcademis ocupa 19.2KB, cerca de metade do tamanho total do *middleware*. As fábricas de componentes ocupam cerca de 7.9KB, sendo suas interfaces definidas no pacote `arcademis` e suas implementações no pacote `rme`. Foi desenvolvida uma versão de RME desprovida de fábricas, na qual cada componente é criado diretamente. De acordo com testes realizados, a criação direta dos componentes do sistema não melhora substancialmente a eficiência de RME, mas contribui para reduzir o tamanho de suas bibliotecas.

5.3 Projeto de *middleware* Reconfigurável em Arcademis.

Conforme discutido no capítulo 3, Arcademis é formado por um núcleo básico de componentes comuns a toda plataforma de *middleware* dele obtida e por um conjunto de objetos acoplados a tal núcleo. Instâncias de Arcademis são formadas por componentes que estendem as classes e interfaces que compõem o núcleo básico e por alguns componentes concretos que fazem parte da lista de funcionalidades disponibilizada pelo arcabouço.

A reconfiguração de instâncias de Arcademis envolve a alteração de um ou mais de seus componentes concretos. Por exemplo, para modificar o protocolo de transporte de dados utilizado em RME, de TCP para UDP, basta substituir a fábrica de canais por uma outra implementação que crie canais de comunicação baseados no novo protocolo. Na implementação de RME, tal alteração pode ser feita sem que seja necessário alterar outros módulos que compõem o sistema, porque todo o código deste *middleware* utiliza a interface de canal fornecida por Arcademis. Assim, caso o novo componente defina todos os métodos desta interface, a maneira como ele é implementado não interfere nas demais partes do projeto.

O exemplo anteriormente apresentado ilustra uma recomendação de projeto que deve ser observada a fim de que as instâncias derivadas de Arcademis possam ser facilmente reconfiguradas:

```
public class BadServiceHandler extends ServiceHandler {
  public void open(Channel ch) {
    try {
      ((BadZipChannel)ch).sendCompressedData("Mensagem".getBytes());
      String s = new String(((BadZipChannel)ch).recvCompressedData());
      System.out.println(s);
    } catch (NetworkException e) {}
  }
}
```

Figura 5.1: Exemplo de implementação pouco flexível.

a programação de sistemas de *middleware* derivados de Arcademis deve basear-se nas interfaces fornecidas por este arcabouço, devendo o desenvolvedor restringir ao mínimo indispensável a alteração das mesmas.

A fim de melhor ilustrar a importância da recomendação anterior, pode-se considerar a implementação de processador de serviço fornecida na Figura 5.1. Neste exemplo, é feita uma coerção explícita sobre o tipo do canal de comunicação, a fim de que a interface fornecida pela classe `BadZipChannel` seja utilizada para transmissão e recepção de dados em vez da interface definida por Arcademis para o componente `Channel`. A implementação mostrada pode atender aos interesses de seu desenvolvedor em determinado momento; porém, caso ele decida alterar o tipo de canal utilizado, não poderá fazê-lo simplesmente modificando a fábrica deste componente, pois pode ser que o novo canal não apresente os métodos `sendCompressedData` e `recvCompressedData`, particulares de `BadZipChannel`.

A implementação mostrada na Figura 5.2 apresenta uma alternativa mais flexível àquela vista na Figura 5.1. Neste caso, um decorador de canais é utilizado para prover a funcionalidade deseja ao canal de comunicação, que continua utilizando a interface fornecida por Arcademis. Caso seja necessário alterar a implementação do canal, basta modificar a implementação da fábrica `ExampleChFactory`. Nenhuma outra alteração em todo o código do sistema de *middleware* é necessária caso os seus demais módulos também utilizem somente a interface definida pelo arcabouço.

5.4 Contribuições desta Dissertação.

Esta dissertação trouxe contribuições metodológicas e práticas, as quais são listadas a seguir:

- Descrição da arquitetura de sistemas de *middleware* orientados por objetos. A divisão de tais sistemas em treze partes independentes, conforme mostrado na Seção 3.1, torna explícitos os principais componentes de *middlewares* orientados por objetos e facilita o entendimento de tais sistemas.

- Utilização de padrões de projeto no desenvolvimento de plataformas de *middleware*. Embora outras plataformas de *middleware* façam intenso uso de padrões de projeto em suas

```
public class ExampleChFactory implements ChannelFc {
    public Channel createChannel() {
        TcpSocketChannel tsc = new TcpSocketChannel();
        ZipChannel zc = new ZipChannel(tsc);
        return zc;
    }
}

public class GoodServiceHandler extends ServiceHandler {
  public void open(Channel ch) {
    try {
      ch.send("Mensagem".getBytes());
      String s = new String(ch.recv());
      System.out.println(s);
    } catch (NetworkException e) {}
  }
}
```

Figura 5.2: Exemplo de implementação flexível.

implementações, esta dissertação expôs como padrões pouco empregados no desenvolvimento de sistemas distribuídos como *flyweight* poderiam ser utilizados.

- Descrição do padrão de projeto *request-response*. Este padrão, apresentado na Seção 3.1.6, facilita a configuração da semântica de chamadas remotas, que fica restrita a, no máximo, quatro de seus componentes: os processadores de serviço `RequestSender`, `RequestReceiver`, `ResponseSender` e `ResponseReceiver`.

- Enumeração de possíveis reconfigurações em plataformas de *middleware*. Além de enumerar tais reconfigurações, esta dissertação mostrou como muitas delas poderiam ser realizadas sobre componentes de Arcademis ou de RME, uma instância deste arcabouço.

- Implementação de um arcabouço para o desenvolvimento de plataformas de *middleware* em Java. Conforme discutido na Seção 5.1, Arcademis pode ser utilizado para o desenvolvimento de uma ampla gama de sistemas de *middleware*, principalmente plataformas orientadas por objetos. Além de conter uma descrição sobre como muitos dos componentes de tais plataformas interagem, Arcademis fornece ao desenvolvedor uma série de componentes concretos, como canais de comunicação, filas de prioridade e identificadores que podem ser utilizados para agilizar o processo de desenvolvimento.

- Implementação de RME. O sistema RME fornece para a configuração CLDC de *Java 2 Micro Edition* um serviço de invocação remota de métodos que pode ser utilizado para facilitar o processo de desenvolvimento de aplicações distribuídas para computação móvel. Não existem relatórios de outros sistemas desenvolvidos para CLDC que forneçam serviços semelhantes aos providos por RME.

5.5 Trabalhos Futuros

Diversas linhas de pesquisa diferentes podem ser criadas a partir de Arcademis e de RME. No caso de Arcademis, é desejável que este arcabouço seja utilizado na instanciação de outras plataformas de *middleware*. Em particular, boas contribuições podem resultar da utilização de Arcademis para a derivação de sistemas de *middleware* que não sejam baseados em invocação remota de métodos, tais como PeerSpaces [VBBP02], Lime [CVV01] ou MPI [MPI95].

Outro ponto que pode vir a ser pesquisado é a parametrização de diferentes mecanismos de coleta de lixo distribuída em Arcademis. Existem vários algoritmos diferentes para este fim, para os quais poderia ser definido um conjunto mínimo de funcionalidades que toda plataforma de *middleware* deveria fornecer. A utilização de um ou outro coletor de lixo ficaria a cargo do desenvolvedor de plataformas de *middleware*. Idealmente tais mecanismos deveriam poder ser configurados da mesma forma que os outros componentes de Arcademis: como uma fábrica de componentes associada à classe `ORB`.

Em relação à plataforma RME, pretende-se melhorar a eficiência do sistema, de modo que a versão do mesmo para J2SE possua um desempenho igual ou melhor que Java RMI. Além disto, RME pode também ser otimizado em relação ao tamanho de suas bibliotecas. Por fim, é desejável que o sistema possa ser testado em um dispositivo real, pois todos os testes realizados durante a implementação deste *middleware* foram realizados em *softwares* que emulam aparelhos celulares e *palmtops*.

Ainda em relação a RME, diversas variantes do sistema podem ser desenvolvidas. Por exemplo, pode-se implementar um serviço de chamadas remotas de método dotado de maior tolerância a falhas. Uma possível solução, neste caso, seria prover *stubs* com uma lista de referências remotas: uma referência padrão e outras que seriam usadas quando o primeiro servidor não estivesse disponível. Outra otimização que poderia ser realizada sobre RME é a utilização de um *buffer* para armazenar mensagens, tal qual descrito na Seção A.7, com o propósito de melhorar a utilização da banda de transmissão de dados.

Apêndice A

Exemplos de Reconfiguração em Arcademis

Este apêndice apresenta uma série de exemplos que mostram como Arcademis e a plataforma RME podem ser reconfigurados de modo a tratar diferentes requisitos. Conforme discutido na Seção 2.5.1, arcabouços para o desenvolvimento de *software* podem ser divididos em duas classes: os sistemas do tipo "caixa-preta" e os do tipo "caixa-branca". Arcademis apresenta características de ambos os tipos de arcabouços. Essencialmente o sistema é composto por um conjunto de classes abstratas para as quais devem ser fornecidas implementações. Por outro lado, Arcademis também disponibiliza aos desenvolvedores de aplicações uma série de componentes já implementados e que podem ser combinados de forma relativamente ortogonal. Isto pode ser feito sem que seja necessário utilizar classes externas, isto é, que não pertencem ao arcabouço.

A título de exemplo, Arcademis disponibiliza duas implementações diferentes de canais de comunicação. Uma destas utiliza um serviço de transmissão orientado por conexão, pois baseia-se no protocolo TCP. A outra destas implementações, por sua vez, utiliza um serviço de transmissão baseado em pacotes de dados (*datagramas*), tendo sido construída sobre o protocolo UDP. Dada uma instância do arcabouço, a modificação do canal de comunicação utilizado é trivial, pois resume-se a alterar a fábrica responsável pela criação deste componente.

Conforme anteriormente dito, Arcademis também pode ser utilizado como um sistema do tipo "caixa-branca". Por exemplo, a classe `RemoteObject` é abstrata, e não fornece implementação para os métodos `activate` e `deactivate`. Estes devem ser implementados em toda instância de *middleware* derivada de Arcademis. Por outro lado, em `RemoteObject` estão implementados métodos tais como `equals` ou `toString()`, que definem alguns aspectos semânticos de objetos distribuídos. Caso seja necessário modificar tal semântica, o desenvolvedor de *middlewares* pode alterar a implementação de `RemoteObject` ou estender esta classe sobrescrevendo os métodos que julgar necessário.

A.1 Reconfiguração de Canais via Decoradores

Em linguagens orientadas por objetos, como Java, por exemplo, dada uma classe, a herança é o mecanismo mais natural para adicionar funcionalidades a ela. Seja, por exemplo, a classe `Channel`, que representa, em Arcademis, um canal de comunicação. A fim de adicionar um *buffer* a esta classe, um caminho possível é estendê-la com uma classe que implementa esta funcionalidade e sobrescrever os métodos necessários. Assim, antes de enviar qualquer *byte*, este é armazenado no *buffer* e somente quando a capacidade do mesmo é atingida, as informações são transmitidas. Continuando este exemplo, seja `BufferedChannel` o nome dado à classe que estende `Channel` agregando-lhe um *buffer*. Para adicionar às mensagens transmitidas pelo canal um sistema de correção como o código Hamming [MS77], poder-se-ia estender a classe `BufferedChannel` e adicionar-lhe as novas funcionalidades. Como é possível perceber, a cadeia de heranças pode tornar-se longa e complexa. Além disso, este mecanismo não é flexível, pois não possibilita que o conjunto de funcionalidades que farão parte do canal seja utilizado de forma ortogonal, isto é, não permite que capacidades sejam inseridas ou removidas de forma independente umas das outras.

Uma abordagem mais flexível que a descrita acima é fornecida pelo padrão de projeto conhecido como *decorator* [GHJV94, Coo00]. Este padrão fornece uma maneira de modificar o comportamento de objetos sem, contudo, levar à criação de longas hierarquias de classes. Assim, um decorador para um canal é uma classe que, em primeiro lugar, é ela própria um canal, isto é, estende a classe `Channel`. Em segundo lugar, um decorador de canal possui um atributo do tipo `Channel`, o que o torna também uma classe cliente daquele componente.

Cada decorador sobrescreve os métodos da classe decorada de modo a agregar-lhe as funcionalidades desejadas. Quando um método é invocado sobre um decorador de canal, os parâmetros do mesmo podem ser alterados antes de serem repassados para o componente decorado. Como decoradores de canais recebem como parâmetro, no instante de sua criação, um objeto do tipo `Channel`, e como possuem eles próprios este tipo, tais entidades podem ser agrupadas de forma independente, ou seja, no caso de canais, diversas funcionalidades diferentes podem ser aglutinadas via composição de decoradores.

Arcademis fornece ao usuário uma classe denominada `ChannelDecorator` que é a classe ancestral de todo decorador de canal. A relação entre a interface `Channel` e a classe `ChannelDecorator`, além das operações definidas em ambos os componentes é mostrada na Figura A.1. Observe que, além de implementar a interface `Channel`, o decorador também possui um atributo desse tipo, conforme explicado anteriormente.

O código da classe `ChannelDecorator` é bastante simples. Cada um dos seus métodos que sobrescreve um método da interface `Channel` apenas retransmite para o canal os parâmetros da operação. Um trecho deste código é mostrado na Figura A.2 (a). A fim de ilustrar o uso do padrão de projeto *decorator*, nesta seção é apresentado um exemplo de decorador simples, que somente imprime na saída padrão o tamanho de cada mensagem enviada. Tal decorador precisa

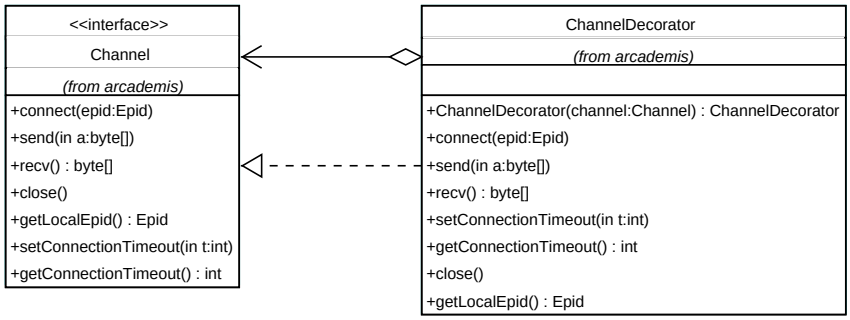

Figura A.1: Relação entre a interface `Channel` e a classe `ChannelDecorator`.

```
public class ChannelDecorator
implements Channel {
  protected Channel channel = null;
  public ChannelDecorator(Channel ch){
    this.channel = ch;
  }
  public void connect(Epid epid)
  throws NetworkException {
    channel.connect(epid);
  }
  // other interceptors here
  // ...
  public void send(byte[] b)
  throws NetworkException {
    channel.send(b);
  }
}
```

(a)

```
public class PrintMsgSizeDecorator
extends ChannelDecorator{
  public PrintMsgSizeDecorator(Channel c){
    super(c);
  }
  public void send(byte[] b)
  throws NetworkException {
    System.out.println("> " + b.length);
    super.channel.send(b);
  }
  public byte[] recv()
  throws NetworkException {
    byte b[] = super.channel.recv();
    System.out.println("> " + b.length);
    return b;
  }
}
```

(b)

Figura A.2: (a) Classe `ChannelDecorator` (b) Exemplo de uso.

modificar somente o corpo dos métodos `send` e `recv` conforme mostrado na Figura A.2 (b).

Arcademis prevê alguns tipos de decoradores que podem ser combinados de diferentes formas a fim de agregar funcionalidades a canais de comunicação. A Tabela A.1 mostra os principais decoradores definidos por este arcabouço.

Conforme anteriormente discutido, uma das vantagens do padrão *decorator* é a possibilidade de utilização de diversos decoradores em conjunto, sem que isto leve à criação de complexas cadeias hierárquicas. Além disso, decoradores bem implementados podem ser agrupados sem que qualquer ordenamento entre eles seja necessário. A Figura A.3 mostra um exemplo de composição de decoradores. Neste exemplo, três capacidades são adicionadas a um mesmo canal. A ordem em que os decoradores são criados pode ser modificada sem que o código interno de qualquer dos componentes precise ser alterado, o que não seria possível caso tais funcionalidades fossem adicionadas via herança. A ordem em que os decoradores são agrupados é definida na fábrica de canais, logo, para alterá-la, seja adicionando ou removendo componentes, basta modificar a fábrica utilizada.

`BufferedChannel`	Adiciona um *buffer* ao canal de modo a tornar mais eficiente a transmissão de *bytes*.
`LineChannel`	Permite transmitir mensagens como linhas de texto, isto é, cadeias de caracteres seguidas por '`\n`'.
`CheckedChannel`	armazena a quantidade de *bytes* lidos de modo a retransmitir apenas as informações não enviadas caso alguma falha ocorra durante a transmissão de dados.
`DataChannel`	Permite tratar cadeias de *bytes* como valores de determinado tipo, por exemplo `int` ou `boolean`. A implementação desse tipo de decorador depende da versão da plataforma Java que estiver sendo utilizada. Por exemplo, J2ME não suporta o tipo ponto flutuante.
`ZipChannel`	Compacta as mensagens antes de enviá-las
`PushbackChannel`	Fornece um *buffer* a partir do qual operações podem ser desfeitas. Por exemplo, após enviar alguns dados, o usuário do canal pode decidir que a informação não deveria ter sido transmitida, e assim, anulá-la.

Tabela A.1: Alguns tipos de decoradores previstos na especificação de Arcademis.

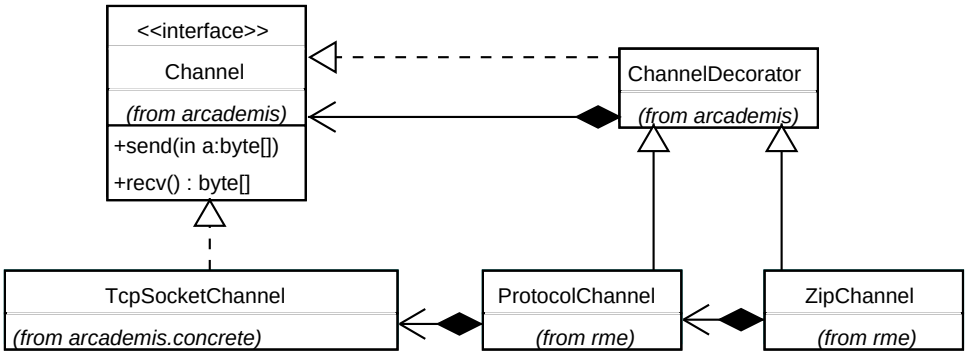

Figura A.3: Exemplo de Composição de Decoradores.

```
package arcademis.concrete;
import arcademis.*;
public class AssynchronousConnector extends Connector implements Runnable {
  private Channel ch = null;private ServiceHandler sh = null;
  private Epid epid = null;private boolean finished = false;
  public void connect(ServiceHandler sh, Epid epid) throws NetworkException {
    this.sh = sh;
    this.epid = epid;
    Thread t = new Thread(this);
    t.start();
  }
  public void run() {
    while(finished == false) {
      try {
        tryConnectionEstablishment();
      } catch(NetworkException ne) {
        try {Thread.sleep(1000);}catch(InterruptedException ie) {}
      }
    }
  }
  private void tryConnectionEstablishment() throws NetworkException {
    if(ch == null)
      ch = OrbAccessor.getChannel();
      ch.connect(epid);
      finished = true;
      sh.open(ch);
    }
}
```

Figura A.4: Conector assíncrono.

A.2 Estabelecimento de Conexões

A implementação de `Connector` utilizada por RME procura estabelecer conexões sincronamente. Assim, a aplicação permanece bloqueada durante o processo de estabelecimento de conexão, e, caso não seja possível a efetivação da mesma, uma exceção é disparada. Diversos motivos podem levar à impossibilidade da realização de uma conexão. Por exemplo, em computação móvel, nodos podem tornar-se inalcançáveis, ficando as aplicações executadas sobre os mesmos incapazes de efetuar conexões com outros elementos do sistema distribuído.

A implementação de `Connector` mostrada na Figura A.4 não causa o bloqueio da aplicação durante o processo de efetivação de conexão. A implementação mostrada procura realizar, em intervalos regulares de 1 segundo, sucessivas tentativas de estabelecimento de conexão, e apenas interrompe este processo quando um canal puder ser construído com o receptor corresponde. Ante a ocorrência de falhas no processo de estabelecimento de conexão, uma exceção do tipo `NetworkException` é disparada, o que causa uma interrupção da *thread* corrente por um segundo.

A.3 Despacho de Requisições

Conforme discutido no Capítulo 3, a estratégia de despacho de requisições define como chamadas remotas são passadas pelo *request receiver* para a implementação do objeto remoto responsável por processa-las. Dado o uso de decoradores, é possível agregar novas funcionalidades

```
package rme.extras.server;
import rme.*;import arcademis.*;import rme.server.*;import arcademis.server.*;
public class ReportLoadDispatcher extends DispatcherDecorator {
  private long oldTime = 0; private double oldAv = .0; private int oldN = 0;
  public ReportLoadDispatcher(Dispatcher dispatcher) {
    super(dispatcher);
    this.oldTime = System.currentTimeMillis();
    this.oldAv = .0;
    this.oldN = 0;
  }
  public Stream dispatch(RemoteCall c) {
    long load = this.updateLoad();
    if(c instanceof RmeRemoteCall)
      if( ((RmeRemoteCall)c).getOperationCode() < 0 ) {
        Stream returnValue = OrbAccessor.getStream();
        try {returnValue.write(load);} catch (MarshalException e) {}
        return returnValue;
    }
    return super.dispatcher.dispatch(c);
  }
  private long updateLoad() {
    double newAv = .0;
    long currentTime = System.currentTimeMillis();
    long timeInterval = currentTime - this.oldTime;
    this.oldAv = (this.oldAv * this.oldN + timeInterval) / (this.oldN + 1);
    this.oldTime = currentTime;
    this.oldN++;
    return (long)this.oldAv;
  }
}
```

Figura A.5: Exemplo de decorador de *Dispatcher*.

ao *Dispatcher*, de um modo semelhante ao utilizado para agregar capacidades extra a canais de comunicação. Nesta seção é mostrado como um decorador de *Dispatcher* pode ser utilizado para informar às aplicações clientes sobre a taxa de utilização do servidor.

Caso uma aplicação cliente possa escolher entre diversos servidores antes de efetuar uma invocação remota, então ela pode optar por aquele que tem recebido uma quantidade menor de requisições ao longo de sua história, pois é provável que tal servidor possa processar a invocação remota mais rapidamente. Com este fim, é necessário que a aplicação servidora tenha como monitorar a quantidade de chamadas recebidas e possa informar aos clientes este dado. O decorador mostrado na Figura A.5 presta-se justamente a este fim.

O decorador mostrado na Figura A.5 utiliza a Equação A.1 para estimar a taxa de recebimento de requisições. Nesta equação, o símbolo m' denota o novo intervalo médio entre o recebimento de chamadas sucessivas e o símbolo m denota a média obtida até o momento. O símbolo t' denota o instante de tempo atual, ao passo que t representa o momento em que a última invocação remota foi recebida. Por fim, o valor n representa a quantidade de invocações remotas recebidas até o instante t. Esta fórmula não atribui um peso maior à história recente do servidor, pois trata-se de um exemplo simples. Uma fórmula mais completa deveria considerar de forma diferente os intervalos obtidos entre as primeiras chamadas tratadas pelo servidor e as últimas.

$$m' = \frac{m \times n + (t' - t)}{n + 1} \tag{A.1}$$

Sempre que o decorador mostrado na Figura A.5 recebe uma requisição cujo código de operação é inferior a zero, ele não a repassa para o próximo *Dispatcher* na cadeia de decoradores. Em vez disto, ele retorna um número serializado que denota a média de tempo entre o processamento de sucessivas requisições desde que o servidor foi inicializado. Esta informação pode ser repassada para o cliente de diversas formas. Na Seção A.4 é mostrado como o protocolo do *middleware* pode ser alterado a fim de que a carga do servidor possa ser informada para o cliente.

A.4 Protocolo do *Middleware*

Para que o decorador de `Dispatcher` que monitora a utilização do servidor possa vir a ser útil para aplicações clientes, é necessário que estas tenham como obter as informações coletadas. O cliente pode obter tais informações enviando invocações remotas cujo código da operação seja inferior a zero. O resultado gerado por tais chamadas é inversamente proporcional à carga sobre o servidor. Contudo, existem outros modos pelos quais a carga sobre o servidor pode ser informada para aplicações clientes. Nesta seção é mostrado como o protocolo do *middleware* pode ser modificado a fim de permitir que tal informação seja transmitida.

Conforme discutido na Seção 3.1.5, o protocolo de comunicação utilizado por qualquer instância de Arcademis é definido por um conjunto de classes que implementam `Message`, pela classe `arcademis.Protocol` e por um conjunto de processadores de serviço que determinam a máquina de estados que caracteriza o protocolo. Assim sendo, alterações no protocolo de comunicação podem ser realizadas mediante alterações nestes três grupos de componentes.

Para que clientes possam inquirir servidores sobre suas taxas de utilização, o protocolo RMEP será acrescido com duas mensagens: *inq* e *load*. A primeira mensagem é enviada pelo cliente até o servidor, e o leva a informar sua taxa de utilização. Esta é enviada em uma mensagem do tipo *load*. O código da classe `InqMsg` é mostrado na Figura A.6 (a) e o código da classe `LoadMsg` pode ser visto na Figura A.6 (b).

Neste exemplo, a aplicação cliente envia mensagens do tipo *inq* para o servidor por meio de um processador de serviços próprio, instância da classe `CheckLoad`. O método `open` de tal classe pode ser visto na Figura A.7 (a). Do lado servidor foi necessário acrescer o código de *request receiver* com uma cláusula específica para mensagens do tipo *inq*. O método `open` deste processador de serviço é mostrado na Figura A.7 (b).

```
package rme.rmep;

import arcademis.*;

public class InqMsg
implements Message {
  public void marshal(Stream b)
  throws MarshalException {
    // send header: RMEP
    b.write(0x524D4550);
    // send protocol version:
    b.write((byte)0x01);
  }

  public void unmarshal(Stream b)
  throws MarshalException {
    int header = b.readInt();
    if(header != 0x524D4550)
      throw new MarshalException();

    byte version = b.readByte();
    if(version != (byte)0x01)
      throw new MarshalException();
  }
}
```

(a)

```
package rme.extras.inquire;
import arcademis.*;
public class LoadMsg
implements Message {
  Stream load = null;
  public void setLoad(Stream l){
    load = l;
  }
  public Stream getLoad(Stream l){
    return this.load;
  }
  public void marshal(Stream b)
  throws MarshalException {
    // send header: RMEP
    b.write(0x524D4550);
    // send protocol version:
    b.write((byte)0x01);
    // send load value
    b.append(load);
  }
  public void unmarshal(Stream b)
  throws MarshalException {
    int header = b.readInt();
    if(header != 0x524D4550)
      throw new MarshalException();
    byte version = b.readByte();
    if(version != (byte)0x01)
      throw new MarshalException();
    this.load = b;
  }
}
```

(b)

Figura A.6: Implementação das mensagens (a) *inq* e (b) *load*.

```
public void open(Channel ch) {
  try {
    super.protocol.setChannel(ch);
    // creating and sending the inquiry
    InqMsg inq = new InqMsg();
    super.protocol.send(inq);
    // receiving the response
    Message msg = super.protocol.recv();
    if(!(msg instanceof LoadMsg))
      throw new ProtocolException();
    LoadMsg lMsg = (LoadMsg)msg;
    Stream returnValue = lMsg.getLoad();
    this.load = returnValue.readLong();
  } catch (Exception e) {}
}
```

(a)

```
public void open(Channel ch) {
  super.protocol.setChannel(ch);
  Message msg = super.protocol.recv();
  if(msg instanceof CallMsg) {
    // ........
  } else if (msg instanceof PingMsg) {
    // ........
  } else if (msg instanceof InqMsg) {
    RmeRemoteCall r =
    new RmeRemoteCall(null, null, -1);
    Stream load = dispatcher.dispatch(r);
    LoadMsg l = new LoadMsg();
    l.setLoad(load);
    super.protocol.send(l);
  } else {
    // handle protocol error here
  }
}
```

(b)

Figura A.7: (a) Método `open` de `CheckLoad`. (b) Método `open` de `RequestReceiver`.

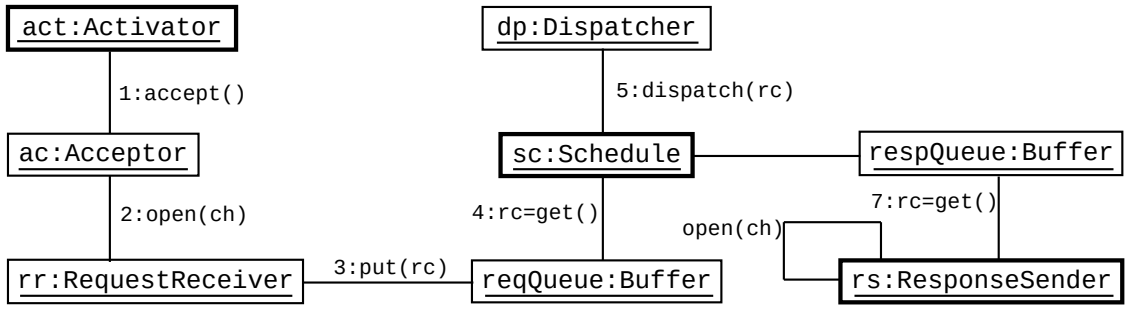

Figura A.8: Servidor baseado em três *threads*.

A.5 Política de *Threads* Adotada pelo Servidor

Um servidor pode utilizar diferentes políticas de *threads* para o processamento de invocações remotas. Diversos dentre os componentes que integram o servidor podem ser executados em *threads* separadas. Por exemplo, na implementação de RME, os componentes `Activator` e `RmeRequestReceiver` são objetos ativos, isto é, executam em um fluxo próprio. `Activator` cria uma *thread* para receber conexões, a fim de permitir que vários objetos remotos possam existir na mesma máquina virtual. `RmeRequestReceiver` inicia uma *thread* para permitir que cada conexão com um cliente seja processada de forma independente.

Arcademis permite que outras políticas, diferentes daquela empregada por RME sejam adotadas. A fim de ilustrar tal fato, esta seção apresenta uma implementação de servidor que utiliza somente três *threads* para processar todas as invocações remotas recebidas. Reduzir o número de fluxos de execução em uma aplicação pode ser vantajoso quando o tempo gasto para a troca de contexto entre *threads* é relevante em relação ao tempo gasto para o processamento das chamadas.

Conforme anunciado anteriormente, o servidor proposto nesta seção utiliza três *threads* para o processamento de invocações remotas, sendo que os principais objetos envolvidos em sua arquitetura são mostrados na Figura A.8. Cada uma destas *threads* é mantida por um componente diferente de Arcademis. Conforme pode ser observado na Figura A.8, os objetos ativos são instâncias de `Activator`, `Scheduler` e `ResponseSender`. A instância de `Activator`, denominada `act`, se ocupa de receber conexões e inicialiar *requests receivers* para tratá-las. Estes últimos componentes, por sua vez, se encarregam de receber seqüências de *bytes* descrevendo chamadas remotas e de inseri-las em uma fila que, na Figura A.8, é denominada `reqQueue`.

Em Arcademis, chamadas remotas são representadas por objetos do tipo `RemoteCall`. Este é o tipo dos elementos inseridos nas filas `reqQueue` e `respQueue` da Figura A.8. Neste caso, a classe `RemoteCall` foi aumentada com um atributo do tipo `Stream` e outro, do tipo `Channel`, que representam, respectivamente, a resposta obtida após o processamento de uma requisição e o canal que deverá ser usado para que tal valor possa ser enviado para a aplicação cliente. Tais

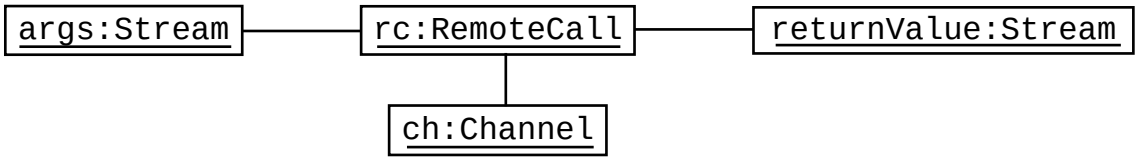

Figura A.9: Implementação da interface `RemoteCall` com novos atributos.

atributos permitem que a classe `RemoteCall` seja utilizada como um elemento de ligação entre a implementação de *request sender* e *request receiver*. A versão de `RemoteCall` utilizada nesta implementação de Arcademis pode ser vista na Figura A.9.

O componente `sc`, instância de `Scheduler`, é executado em outra `thread`, e sua função é enviar para o `Dispatcher` as chamadas remotas inseridas na fila `ReqQueue`. Além disto, o `Scheduler` insere na fila de respostas as requisições remotas uma vez que seu resultado tenha sido obtido. O último dos componentes ativos é representado por uma instância de `Response-Sender`, e sua função é enviar para aplicações clientes o resultado de invocações remotas. Todos os três objetos ativos mostrados na Figura A.8 são inicializados pelo método `activate`, do componente `Activator`.

A.6 Uso de *Caches* em Invocadores

Existem métodos que sempre retornam o mesmo valor quando executados com os mesmos parâmetros. Embora a princípio esta pareça ser uma característica desejável para todos os métodos, existem situações onde ela não é interessante. Por exemplo, uma função que informa números aleatórios deveria retornar um valor diferente a cada invocação. Os métodos que retornam sempre o mesmo valor quando executados com os mesmos parâmetros e que, além disso, não causam efeitos colaterais, são conhecidos como *métodos idempotentes* [CDK96].

Quando métodos idempotentes são executados remotamente, é possível utilizar *caches* para aumentar a eficiência da aplicação cliente. O *cache*, neste caso, consiste em uma tabela onde, para cada método idempotente, são associados os parâmetros do mesmo e o valor de retorno obtido após sua execução. Embora o *cache* custe à aplicação cliente espaço, em termos de memória, ele evita que um novo acesso à rede aconteça quando o mesmo método é invocado com os mesmos argumentos. Em RME, para que aplicações possam usufruir do benefício de um *cache*, basta adicionar um novo decorador à cadeia de invocadores. Na Figura A.10 pode ser visto um exemplo de decorador de `Invoker` que utiliza uma tabela *hash* como implementação de *cache*.

Por ser um exemplo simples, o invocador mostrado na Figura A.10 não verifica se a chamada remota recebida se destina a um método idempotente ou não. Para qualquer chamada, são executados os seguintes procedimentos: o decorador verifica se os argumentos daquela chamada

```
package rme.extras;
import java.util.*; import arcademis.*; import rme.*;
public class CachedInvoker extends InvokerDecorator {
  private Hashtable table = null;
  public CachedInvoker(Invoker invoker) {
    super(invoker);
    table = new Hashtable();
  }
  public Stream invoke(RemoteCall c) throws NetworkException {
    Stream args = ((RmeRemoteCall)c).getArguments();
    if(table.containsKey(args)) {
      Stream returnValue = OrbAccessor.getStream();
      returnValue.fill((RmeStream)table.get(args));
      return returnValue;
    }
    else {
      Stream returnValue = super.invoker.invoke(c);
      Stream backupValue = OrbAccessor.getStream();
      backupValue.fill(returnValue);
      table.put(args, backupValue);
      return returnValue;
    }
  }
}
```

Figura A.10: Decorador que agrega um *cache* à cadeia de invocadores.

já se encontram inseridos na tabela *hash*. Em caso afirmativo, o objeto associado àquela chave é retornado, sem que qualquer acesso à rede subjacente aconteça. Caso a chamada ainda não tenha sido realizada, então, após o seu processamento, seus argumentos são inseridos na tabela, como chave de pesquisa, em conjunto com o valor de retorno obtido após sua execução; em seguida, o valor de retorno obtido é passado para a aplicação cliente.

A.7 Enfileiramente de Requisições Remotas

A fim de diminuir o número de acessos à camada de transporte, é possível agrupar diversas chamadas em uma única massa de dados de modo a transmiti-las, dessa forma aglutinadas, via uma única conexão com o servidor remoto. Desta estratégia podem resultar benefícios para aplicações executadas sobre uma rede lenta [YK03] ou para aplicações que realizam muitas chamadas que não demandam confirmação [SLM98].

O agrupamento de invocações remotas leva a um melhor aproveitamento da rede, pois faz com que uma quantidade menor de informações sejam transmitidas. Por exemplo, caso o protocolo TCP/IP seja utilizado para permitir a comunicação distribuída, então uma série de informações adicionais tais como cabeçalhos e identificadores são adicionados à seqüência de *bytes* que descreve a invocação remota [Tan03], aumentando-se assim a quantidade de informação transmitida. Além disto, cada transmissão de dados causa a execução de uma série de comandos primitivos do sistema operacional, sendo que algumas destas operações são ordens de grandeza mais lentas que a maior parte das instruções utilizadas no processamento dos métodos remotos. Assim, Embora a estratégia de enfileiramento descrita tenda a aumentar o tempo gasto com transmissão

de mensagens, uma vez que elas não são logo transmitidas pelo *stub* tão logo este as receba, é possível que o número de invocações remotas realizadas por unidade de tempo (*throughput*) aumente.

Em Arcademis, para permitir o enfileiramento de chamadas remotas, é preciso adicionar um decorador de `Invoker` à cadeia de decoradores. Tal decorador não repassa as chamadas recebidas diretamente para a aplicação servidora. Ao contrário, ele as insere em uma fila, e, quando determinado parâmetro é atingido, todas as mensagens enfileiradas são enviadas para seu destino em um único bloco de dados. Além de adicionar um novo decorador à cadeia de invocadores, é preciso modificar o protocolo do *middleware* de modo que uma mensagem passe a conter diversas ordens de invocação remota. Por conseguinte, é necessário criar uma nova implementação para `arcademis.Message` e inserir uma cláusula no código do *request receiver* para tratar a nova mensagem.

O conteúdo da fila de mensagens pode ser enviado para o servidor segundo diversas estratégias. Algumas condições que podem determinar a transmissão de mensagens são listadas a seguir. Tais estratégias não são exclusivas, isto é, podem ser combinadas de acordo com as necessidades da aplicação. É importante observar que, dependendo da estratégia de enfileiramento utilizada, chamadas remotas que causam o bloqueio da aplicação cliente não podem ser enfileiradas, devendo ser transmitidas imediatamente. O problema, neste caso, é que, se a fila de mensagens for transmitida somente quando um determinado número de requisições enfileiradas é alcançado, então a aplicação pode permanecer bloqueada indefinidamente.

- **Contador de mensagens**: quando a quantidade de mensagens enfileiradas atinge um certo limite, elas são transmitidas para o servidor.
- **Contador de** *bytes*: quando a soma do tamanho das mensagens enfileiradas atinge um determinado valor, elas são transmitidas. O tamanho de uma mensagem, neste caso, é definido como a quantidade de *bytes* utilizados para representá-la.
- **Período de tempo**: em intervalos regulares de tempo todas as mensagens armazenadas são enviadas para o servidor remoto. Esta solução permite que o ritmo de transmissão de mensagens seja constante, ainda que estas sejam produzidas em intervalos irregulares.
- **Transmissão explícita**: esta opção permite que o usuário intencionalmente cause a transmissão das mensagens enfileiradas.
- **Entrega direta**: esta opção permite que o usuário, sempre que houver necessidade, envie chamadas remotas diretamente para o servidor, isto é, sem enfileira-las em um *buffer*.

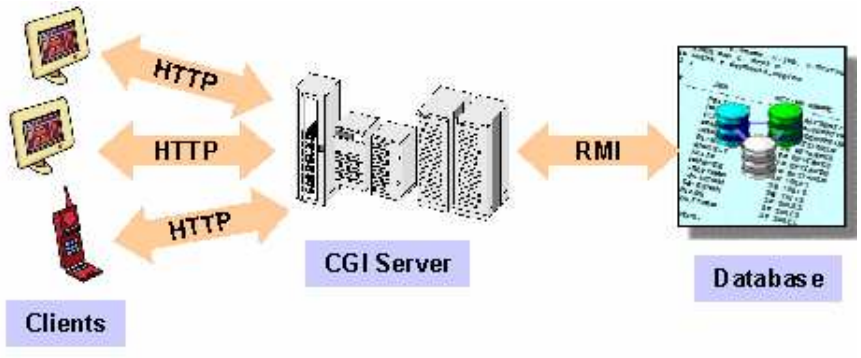

Figura A.11: Exemplo de aplicação para comércio eletrônico.

A.8 Reconfiguração da Agência de Localização

A fim de mostrar como a agência de localização utilizada por uma instância de Arcademis pode ser modificada, nesta seção é mostrado como alterações sobre o seu código permitem criar *stubs* segundo o padrão de projetos denominado *flyweight* [GHJV94, Coo00], o qual pode ser utilizado para reduzir o número de instâncias de algumas classes em determinados contextos. Embora pareça, a princípio, que cada classe implementada segundo tal padrão seja um *singleton*, não é este o caso, pois o padrão *flyweight* permite que exista mais de uma instância para a mesma classe. Este padrão busca compartilhar instâncias entre diferentes referências sempre que isto for possível. Por exemplo, caso duas referências remotas denotem o mesmo objeto, então elas podem compartilhar o mesmo *stub*.

Existem aplicações distribuídas em que, vez ou outra, é necessário criar um número grande de referências para o mesmo objeto remoto. Um exemplo típico deste tipo de cenário é uma aplicação para comércio eletrônico em que um cliente pode precisar criar várias conexões com o mesmo banco de dados. A Figura A.11 representa tal situação. Na aplicação mostrada, um servidor CGI recebe formulários contento listas de requisições e, para cada um deles, cria uma conexão com o banco de dados, o qual está localizado em outro *host*. Cada conexão é manipulada por um *stub* que se encontra no espaço de endereçamento do servidor CGI.

O principal problema do projeto mostrado na Figura A.11 é a possibilidade de um número excessivamente grande de *stubs* poderem ser criados no espaço de endereçamento do servidor CGI, sendo que todos eles referenciam a mesma entidade remota, neste caso, o banco de dados. Dependendo da arquitetura de rede utilizada, a existência de diversos *stubs* para o mesmo componente não é vantajosa, pois os mesmos não são capazes de se comunicarem em paralelo com a implementação do objeto remoto. Assim, ao invés de aumentar a eficiência da aplicação, os *stubs* "redundantes" representam somente um desperdício de recursos, uma vez que apenas um *proxy* seria suficiente para garantir toda a comunicação entre cliente e objeto remoto.

A abordagem descrita acima está presente na implementação de Java RMI. A fim de constatar

```
1: RObj ro1 = (RObj) Naming.lookup("remote object");
2: RObj ro2 = (RObj) Naming.lookup("remote object");
3:
4: System.out.println(ro1.toString());
5: System.out.println(ro2.toString());
6: System.out.println(ro1 == ro2);
```

Figura A.12: Criação de *stubs* diferentes para o mesmo objeto remoto em Java RMI.

tal fato, o programa mostrado na Figura A.12 é suficiente. Embora as duas chamadas sobre o método `toString()` retornem a mesma seqüência de caracteres, as referências são diferentes, pois na linha 6 desta figura é impresso o valor `false`.

É interessante utilizar *flyweights* quando, dado um conjunto de instâncias de uma mesma classe, elas são exatamente iguais, exceto por uns poucos parâmetros. No exemplo mostrado na Figura A.11, toda conexão com o banco de dados difere somente em relação à natureza da requisição realizada. Passando tal requisição como o parâmetro de uma chamada remota de método, o número de *stubs* pode ser igualado à quantidade de bancos de dados remotos, a qual, normalmente, é igual a um.

A decisão de criar uma nova instância para uma determinada classe, ou reaproveitar uma referência existente deve ser decidida pela fábrica do componente em questão. Esta fábrica, normalmente mantém uma lista de referências ativas e, dada a solicitação por um novo objeto, define se este é criado realmente, ou se alguma referência armazenada pode ser retornada no lugar da nova instância. Como as fábricas precisam conhecer quais instâncias já foram criadas, em geral elas são programadas como *singletons*. No caso de uma plataforma de *middleware* orientada por objetos, a fábrica de *stubs* é a agência de localização, que os cria via o método `find`, ou outro método de pesquisa similar, como `lookup`, no caso de Java RMI.

Assim, para permitir que *stubs* sejam criados segundo o padrão *flyweight* deve-se modificar a implementação do método `find` fornecido pela agência de localização, a qual passará a ter acesso a uma lista contendo referências para todos os *stubs* já criados. Arcademis define duas interfaces diferentes para o serviço de localização de nomes, porém em ambas, o método `find` retorna uma referência para um *stub*. A fim de empregar o padrão *flyweight*, a agência de localização deveria ser implementada da seguinte forma: ao receber uma consulta, a parte servidora da agência deve retornar uma referência remota (`RemoteReference`) para sua parte cliente. De posse desta referência, a parte cliente do serviço de nomes verifica se já contém em sua lista interna um *stub* equivalente. Em caso positivo, tal componente é retornado para o cliente como o resultado da pesquisa, senão um novo *stub* é criado, inserido na lista, e retornado para o cliente.

Bibliografia

[ABW98] Sherman Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison-Wesley, 1th edition, 1998.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, 3rd edition, 2000.

[Bak97] S. Baker. *Corba Distributed Objects : Using Orbix*. Addison-Wesley, 1997.

[BHTV03] Luciano Baresi, Reiko Heckel, Sebastian Thone, and Daniel Varro. Modeling and validation of service oriented architectures: Application vs. style. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

[BL98] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *14th Symposium on Operating Systems Principles (SOSP)*, pages 217–230. Software–Practice and Experience, 1993.

[CDK96] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design*, volume 1. Addison-Wesley, 2nd edition, 1996.

[CK02] Fábio M. Costa and Fabio Kon. Novas Tecnologias de Middleware: Rumo à Flexibilização e ao Dinamismo. Simpósio Brasileiro de Redes de Computadores, 2002.

[CKR00] Stefano Campadello, Oskari Koskimies, and Kimmo Raatikainen. Wireless Java RMI. In *1th International Enterprise Distributed Object Computing Conference*, pages 114–123. USENIX Association, 2000.

[CLRS02] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Algoritmos – Teoria e Prática*. Campus, 2nd edition, 2002.

[Coo00] James Cooper. *The Design Patterns Java Companion*. Addison-Wesley, 2000.

[CVV01] Bogdan Carbunar, Marco Túlio Valente, and Jan Vitek. Lime Revisited. In *5th IEEE International Conference on Mobile Agents*, pages 54–69. Springer-Verlag, 2001.

[D. 99] D. Schmidt and M. Fayad and R. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley and Sons, 1999.

[D. 02] D. Schmidt and S. D. Huston. *C++ Network Programming: Mastering Complexity with ACE and Frameworks*, volume 2. Addison-Wesley, 2002.

[Deu89] Peter Deutsch. *Design Reuse and Frameworks in the Smalltalk–80 System*, volume 2, chapter 3, pages 57–72. Biggerstaff and Perlis, 1989.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.

[Fra98] Frantisek Plasil and Michael Stal. An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM. *Software - Concepts and Tools*, 19(1):14–28, 1998.

[Gei01] Kurt Geihs. Middleware Challenges Ahead. *IEEE Computer*, 34(6):24 – 31, 2001.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gro95] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. OMG, 2nd edition, 1995.

[Gro03] Object Management Group. Minimumcorba specification, 2003. http://doc.ece-.uci.edu/CORBA/formal/02-08-01.pdf – última visita: junho de 2003.

[GSI03] Nikolaos Georgantas, Daniele Sacchetti, and Valerie Issarny. Making Middleware Communication Architecture Reconfigurable. Technical Report 0279, INRIA, 2003.

[HGM01] Yongqiang Huang and Hector Garcia-Molina. Exactly-once semantics in a replicated messaging system. In *17th International Conference on Data Engineering (ICDE)*, 2001.

[HJS03] Ewald Geschwinde Hans-Jurgen Schonig. *Mono Kick Start*. Sams, 2003.

[JF88] Ralph Johnson and Brian Foote. Designing reusable software. *Journal of Object Oriented Programming*, 1(2):22–35, 1988.

[Joh97] Ralph Johnson. Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

[JS97] Prashant Jain and Douglas Schmidt. Service configurator: A pattern for dynamic configuration of services. In *3rd Conference on Object-Oriented Technologies and Systems*, pages 209–219. USENIX, 1997.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The art of the metaobject protocol*. MIT Press, 1991.

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26 – 49, 1988.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*. Springer-Verlag, 2000.

[Mac03] Matthew MacDonald. *Microsoft .NET Distributed Applications: Integrating XML Web Services and .NET Remoting*. Microsoft Press, 2003.

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Middleware for Mobile Computing (A Survey). *LNCS*, 2497:20 – 58, 2002.

[Mey97] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[MH00] R. Monson-Haefel. *Enterprise Javabeans*. O'Reilly & Associates, 2000.

[Mic03] Sun Microsystems. Java RMI home page, 2003. http://java.sun.com/j2se/-1.4.1/docs/guide/rmi/spec/rmiTOC.html – última visita: junho de 2003.

[MPI95] MPI Forum. The Message Passing Standard, 1995.

[MS77] F. MacWilliams and N. Sloane. *The Theory of ErrorCorrecting Codes*. North-Holland, 1977.

[Nel91] Greg Nelson. *Systems programming with Modula-3*. Prentice Hall, 1991.

[NRV00] V. Natarajan, S. Reich, and B. Vasudevan. *Programming With Visibroker : A Developer's Guide to Visibroker for Java*. John Wiley & Sons, 2000.

[OH01] Piet Obermeyer and Jonathan Hawkins. Microsoft .NET Remoting: A Technical Overview. Technical Report 013, Microsoft Corporation, 2001.

[OMG99] OMG. CORBA IIOP 2.3.1 Specification. Technical Report 99-10-07, OMG, 1999.

[OSK+00] Carlos O'Ryan, Douglas Schmidt, Fred Kuhns, Marina Spivak, Jeff Parsons Irfan Pyarali, and David L. Levine. Evaluating policies and mechanisms for supporting embedded, real-time applications with corba 3.0. In *Sixth IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, 2000.

[PS98] Irfan Pyarali and Douglas Schmidt. An Overview of the CORBA Portable Object Adapter. *ACM StandardView*, 6, 1998.

[PVBB03] Fernando Magno Quintao Pereira, Marco Túlio Valente, Roberto S. Bigonha, and Mariza A. S. Bigonha. Chamada remota de métodos na plataforma J2ME/CLDC. In *V Workshop de Comunicação sem Fio e Computação Móvel*, pages 157 – 168. SBC, 2003.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[RKC01] Manuel Román, Fabio Kon, and Roy Campbell. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5), 2001.

[RMKC00] Manuel Román, Dennis Mickunas, Fabio Kon, and Roy Campbell. LegORB and Ubiquitous CORBA. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*. Springer-Verlag, 2000.

[Rog97] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[RTV01] Roger Riggs, Antero Taivalsaari, and Mark VandenBrink. *Programming Wireless Devices with the Java 2 Platform, Micro Edition*. Addison Wesley, 1th edition, 2001.

[SB03] Doublas Schmidt and Frank Buschmann. Patterns, frameworks and middleware: Their synergestic relationships. In *25th international conference on Software engineering*, pages 694–704. ACM, 2003.

[SC99] Doublas Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible and Maintainable ORB Middleware. *IEEE communications Magazine – Special Issue on Design Patterns*, 37(4):54 – 63, 1999.

[Sch94] Douglas Schmidt. Reactor – an object behavioral pattern for event demultiplexing and event handler dispatching. In *First Pattern Languages of Programs conference*, 1994.

[Sch96] Douglas Schmidt. Acceptor-connector – an object creational pattern for connecting and initializing communication services. In *European Pattern Language of Programs conference*, 1996.

[SG97] Doublas Schmidt and Andy Gokhale. Evaluating the performance of demultiplexing strategies for real-time CORBA. In *GLOBEGOM'97*. IEEE, 1997.

[SG98] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. John Wiley & Sons, Inc, 5th edition, 1998.

[SHM+00] Bill Shannon, Mark Hapner, Vlada Matena, James Davidson, Eduardo Pelegri-Llopart, and Larry Cable. *Java 2 Platform, Enterprise Edition: Platform and Component Specifications*. Addison Wesley, 1th edition, 2000.

[Sin99] Ashish Singhai. *Quarterware: A Middleware Toolkit of Software RISC Components*. PhD thesis, University of Illinois, 1999.

[SL95] Douglas Schmidt and Greg Lavender. Active object – an object behavioral pattern for concurrent programming. In *Second Pattern Languages of Programs conference*, 1995.

[SLM98] Doublas Schmidt, David Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4), 1998.

[SSC98] Ashish Singhai, Aamod Sane, and Roy H. Campbell. Quarterware for middleware. In *ICDC'98*. IEEE, 1998.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[Ste93] Steve Vinoski. Distributed Object Computing With CORBA. *C++ Report magazine*, 1993.

[Ste98] Richard Stevens. *UNIX Network Programming*, volume 1. Prentice Hall, 2nd edition, 1998.

[STK01] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly & Associates, 2001.

[Tan03] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, 4th edition, 2003.

[VBBP02] Marcos Túlio Valente, Roberto Bigonha, Mariza A. S. Bigonha, and Fernando Magno Quintão Pereira. A Coordination Model for Mobile Ad Hoc Systems and its Formal Semantics. *Brazilian Workshop on Mobile Computing*, 2002.

[Vin98] Steve Vinoski. New Features for CORBA 3.0. In *Communications of the ACM*, pages 44–52. ACM Press, New York, NY, 1998.

[Wel03] Matt Welsh. The Ninja Project, 2003. http://ninja.cs.berkeley.edu/ – última visita: outubro de 2003.

[WRW96] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX Association, 1996.

[WWB+03] A. Wigley, S. Wheelwright, R. Burbidge, R. MacLeod, and M. Sutton. *Microsoft .NET Compact Framework (Core Reference)*. Microsoft Press, 1th edition, 2003.

[WWWK97] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.

[YK03] Kwok Cheung Yeung and Paul Kelly. Optimising Java RMI Programs by Communication Restructuring. In *Lecture Notes in Computer Science*, volume 2672/2003, pages 324–343. Springer-Verlag Heidelberg, 2003.