Chamada Remota de Métodos na Plataforma J2ME/CLDC

Fernando M. Q. Pereira[1] Roberto S. Bigonha[1]
Marco Túlio O. Valente[2] Mariza A. S. Bigonha[1]

[1]Dept. de Ciência da Computação – Universidade Federal de Minas Gerais
[2]Dept. de Ciência da Computação – Pontifícia Universidade Católica de Minas Gerais

Resumo. Este artigo apresenta o sistema *RME*, uma plataforma de *middleware* que fornece aos desenvolvedores de aplicações distribuídas um serviço de invocação remota de métodos. RME foi desenvolvido sobre o perfil MIDP/CLDC da plataforma J2ME, a versão da linguagem Java voltada para dispositivos limitados computacionalmente, como aparelhos celulares e *palmtops*. A implementação de RME, baseada em uma série de conhecidos padrões de projeto, oferece aos desenvolvedores diversas opções de reconfiguração, o que a torna adequada tanto para pequenos dispositivos quanto para computadores dotados de recursos mais poderosos.

Abstract. This paper presents the RME system, a middleware plataform that aims to provide application developers with a remote method invocation service. The platform has been developed to the J2ME version of the Java language, which is directed to small devices like cell phones and palmtops. More specifically, it targets a configuration of J2ME called CLDC. Its implementation uses several design patterns which make it easy to reconfigure the system in order to address different requirements imposed by the execution environment. Because of this approch, RME can be used in limited resource devices as well as in more powerful computers.

1 Introdução

Aplicações voltadas para computadores móveis como PDA's e telefones celulares são cada vez mais demandadas. 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 [13]. 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 poderá 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 prover aos projetistas de aplicações um modelo de desenvolvimento comum, foi desenvolvida a plataforma J2ME [13], um ambiente de execução Java que requer menos de um décimo dos recursos necessários ao sistema Java tradicional, conhecido como J2SE [1].

Dentre as principais 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. 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. Em segundo lugar, dada a grande variedade de dispositivos móveis existentes, portabilidade é essencial, pois permite a reutilização de código entre diferentes processadores e facilita a comunicação entre sistemas distribuídos. 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 será descrito na Seção 2, a plataforma J2ME apresenta diversas configurações, cada uma de-

las específica para um determinado grupo de processadores. A configuração destinada a aparelhos de telefonia celular, denominada CLDC [13], 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.

Java RMI (*Java Remote Method Invocation*) [21], é um modelo de objetos distribuídos desenvolvido para a linguagem Java que pode ser visto como uma evolução do serviço de RPC (*Remote Procedure Call*) [20]. Java RMI 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 permitir que componentes distribuídos possam ser utilizados como se estivessem todos localizados no mesmo espaço de endereçamento, Java RMI disponibiliza aos desenvolvedores de aplicações uma poderosa abstração, pois eles podem se ater aos detalhes do modelo de objetos a ser desenvolvido sem, contudo, preocupar-se com detalhes de comunicação inerentes a uma rede de computadores.

Embora o modelo de invocação remota de métodos adotado nas versões J2SE e J2EE da linguagem Java disponibilize aos seus usuários algumas opções de reconfiguração, sua arquitetura ainda é bastante monolítica. Isto quer dizer que muitos dos componentes do sistema Java RMI não podem ser removidos ou alterados, o que torna o modelo inadequado para plataformas dotadas de poucos recursos, como a configuração CLDC da plataforma J2ME.

O presente trabalho descreve RME (*RMI for J2ME*), um serviço de chamada remota de métodos desenvolvido para a configuração CLDC. O sistema proposto permite que programadores utilizem a mesma sintaxe normalmente utilizada pelo serviço Java RMI presente em outras versões da linguagem. A Seção 2 expõe as principais características da plataforma J2ME. Na Seção 3, por sua vez, é descrita a arquitetura do sistema RME, sendo abordados pontos como os padrões de projeto utilizados em sua implementação e o protocolo de comunicação adotado. Na Seção 4 são enumeradas algumas das plataformas de *middleware* que possuem características semelhantes ao sistema RME e, finalmente, a Seção 5 contém conclusões e descrições de possíveis linhas de pesquisa que podem ser desenvolvidas a partir do trabalho apresentado.

2 A Plataforma J2ME

A linguagem Java possui três distribuições principais, ou edições. Dessas edições, a mais popular é conhecida como J2SE, ou *Java 2 Standard Edition*. A segunda distribuição da linguagem, denominada J2EE [19], 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 dessas 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 essas 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 (PDAs). A segunda destas configurações, denominada CDC (*Connected Device Configuration*), está voltada para dispositivos dotados de maior capacidade computacional, com no mínimo 2 MB de memória disponível [13].

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. Esse perfil foi implementado sobre a configuração CLDC. Também para a configuração CDC foi desenvolvido um perfil, o qual ficou conhecido como *Foundation Profile* [13].

A fim de prover a plataforma J2ME com um ambiente no qual aplicações pudessem ser executadas, foi desenvolvida uma máquina virtual específica, denominada KVM. Esta máquina virtual foi desenvolvida 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 nesse 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 [9].

3 Arquitetura do Sistema Desenvolvido

A arquitetura de RME é baseada em uma série de padrões de projeto, os quais são descrições de soluções normalmente adotadas para alguns problemas comuns em programação [6]. Dentre os padrões utilizados para o desenvolvimento de RME podem-se citar fábricas de objetos, decoradores e *proxies*. Além destes, padrões específicos para sistemas de objetos distribuídos também foram usados, como por exemplo, *reactor* e *acceptor-connector*. O uso de tais padrões será descrito com maiores detalhes nesta e nas próximas seções.

O sistema RME foi desenvolvido com o intuito de poder ser facilmente configurado por desenvolvedores de aplicações de acordo com suas necessidades. A fim de alcançar tais objetivos, muitos dos objetos que compõem a plataforma não são instanciados diretamente, mas a partir de fábricas de objetos, as quais constituem um típico padrão de projeto [6]. Uma fábrica cria uma dentre várias possíveis instâncias de uma classe de acordo com os dados providos para ela. Esse padrão facilita a reconfiguração de RME pois, para alterar um componente do sistema, basta que o desenvolvedor modifique a fábrica responsável por sua criação.

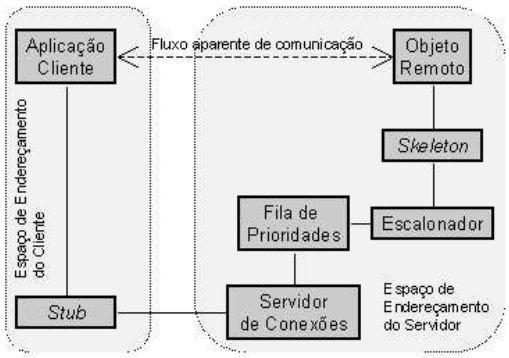

Figura 1: Principais entidades envolvidas na comunicação entre uma aplicação e um objeto remoto

3.1 Comunicação entre Aplicações e Objetos Remotos

Em RME, um objeto cliente utiliza representações de objetos remotos para realizar chamadas de métodos sobre eles. Tais representações locais, ou *proxies* [6], são denominadas *stubs* [9] em Java RMI. Este mesmo nome é utilizado em RME. Um *stub* possui a mesma interface que o objeto remoto que ele representa. Cada um de seus métodos se encarrega de converter os argumentos passados em uma seqüência de *bytes* e enviá-la para uma entidade capaz de executar tais métodos com os parâmetros dados. Além de enviar os parâmetros de uma invocação, o *stub* se encarrega de receber o valor que resulta da execução da mesma.

Um *stub* não se comunica diretamente com o objeto remoto que representa. Na realidade, tanto em Java RMI quanto no sistema RME, o código de objetos que serão utilizados remotamente não precisa ser diferente do código de objetos que somente serão utilizados localmente. Assim, para permitir a comunicação entre um objeto remoto e o correspondente *stub*, no modelo RME existem três intermediários entre estas duas entidades, ambos localizados no espaço de endereçamento do objeto remoto. O primeiro deles é o servidor responsável por receber as mensagens enviadas pelos *stubs* e inserir tais requisições em uma fila de prioridade. O segundo intermediário é um objeto escalonador, cuja função é retirar as requisições da fila, de acordo com a política de prioridade adotada e repassá-las para o terceiro e último intermediário. Este último trata-se do elemento responsável por permitir a comunicação entre o objeto remoto e o escalonador anteriormente descrito. Tal objeto, tanto em Java RMI quanto em RME, é denominado *skeleton* [9] e sua estrutura pode ser descrita por um padrão de projeto conhecido como adaptador [6]. A Figura 1 descreve as principais entidades envolvidas no processo de comunicação entre uma aplicação e um objeto remoto.

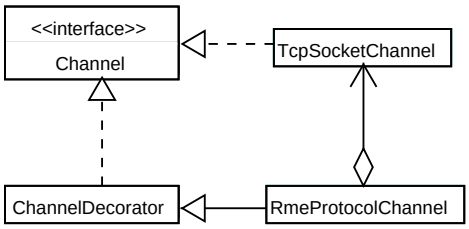

Figura 2: Canal de comunicação e decoradores em RME

O protocolo utilizado em RME denomina-se RMEP e está descrito na Seção 3.7. Para o transporte de seqüências de *bytes* que constituem mensagens desse protocolo foi definida a classe `TcpSocketChannel`. Essa classe, que representa um canal de comunicação, contém métodos para o estabelecimento de conexões e para o envio e recebimento de *bytes*. Para o recebimento de conexões foi definida a classe `TcpSocketServer`.

Caso seja necessário acrescentar funcionalidades adicionais ao canal, como compactação de mensagens, por exemplo, podem ser usadas classes decoradoras. Decoradores [6] constituem um padrão de projeto usado para modificar o comportamento de objetos sem a necessidade de estendê-los via herança. Um decorador pode interceptar chamadas de métodos sobre o objeto decorado, realizar algum processamento sobre essas chamadas baseado em seus parâmetros e repassá-las ao destino original. Para decorar canais de comunicação, RME define a classe `ChannelDecorator`. Esta é a classe ancestral de todos os decoradores de canais, além de ser, ela própria, uma descendente e uma classe cliente de `TcpSocketChannel`. Diversos decoradores, descendentes de `ChannelDecorator`, podem ser agrupados para acrescentar capacidades independentes ao mesmo canal.

A Figura 2 mostra o relacionamento entre o canal de comunicação e o decorador utilizado em RME. A implementação de `TcpSocketChannel` utiliza um *buffer* para tornar mais eficiente o aproveitamento da rede de comunicação, o que permite que somente mensagens completas sejam enviadas. Essa capacidade melhora o aproveitamento do canal, pois pacotes contendo poucos *bytes* deixam de existir. O segundo decorador implementa RMEP, o protocolo descrito na Seção 3.7. Caso seja necessário adicionar outras funcionalidades ao canal de comunicação, o usuário de RME pode definir decoradores adicionais para o mesmo.

Tanto decoradores quanto a extensão via herança permitem agregar novas funcionalidades a componentes já implementados, ou alterar o comportamento dos mesmos. Contudo, decoradores são um mecanismo mais flexível [11], uma vez que vários deles podem ser combinados de diversas formas diferentes, o que não se verifica em hierarquias de classes. Por exemplo, o arranjo de classes mostrado na Figura 2 poderia ser alterado adicionando-se ou removendo-se um decorador, sem haver a necessidade de modificar o código dos demais componentes. Por outro lado, em uma hierarquia de classes na linguagem Java um novo componente somente pode ser inserido via a extensão da classe mais específica.

3.2 Estabelecimento de Conexões

O tratamento de conexões, em RME, utiliza um padrão de projeto denominado *acceptor-connector* [17]. A principal vantagem deste padrão é separar de forma bem definida o estabelecimento de uma conexão do uso da mesma. Uma segunda vantagem é tornar mais simples a realização de chamadas assíncronas, pois esse padrão de projeto permite que a aplicação não seja bloqueada durante o estabelecimento de uma conexão. Tal estabelecimento se dá entre duas entidades, o *conector* e o *receptor*. Conectores representam a parte ativa no processo, uma vez que são responsáveis pelo seu início. Receptores, por sua vez, são entidades passivas, pois executam atividades mediante a chegada de requisições de conexões.

Quando uma requisição de conexão é verificada pelo receptor, um canal é criado e entregue a um objeto responsável por processar as mensagens que serão enviadas por ele. Este componente, denominado *service handler*, ou processador de serviço, é responsável pela troca de mensagens e pelo fechamento do canal, quando o mesmo não for mais necessário. É interessante ressaltar que as duas partes que se comunicam não precisam necessariamente utilizar os mesmos processadores de serviço. Em RME são definidos quatro desses processadores, cuja relação com as demais classes envolvidas neste padrão de projeto é mostrada na Figura 3.

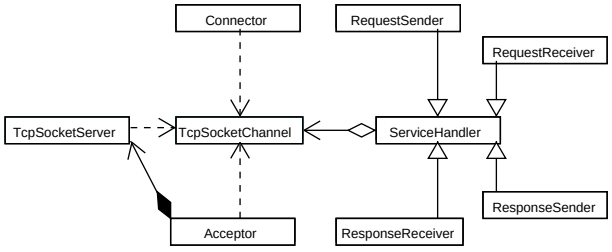

Figura 3: Principais classes envolvidas no estabelecimento de conexões em RME

Para enviar requisições de chamadas remotas para servidores, *stubs* utilizam um processador de serviços denominado `RequestSender`. Servidores, por sua vez, utilizam um processador de serviços denominado `RequestReceiver` para receber chamadas remotas e outro, instância da classe `ResponseSender`, para enviar a resposta. Finalmente, o *stub* utiliza uma instância da classe `ResponseReceiver` para receber o valor de retorno ou confirmação de uma invocação. A relação entre esses quatro elementos é mostrada na Figura 4.

As estruturas apresentadas nessa figura serão discutidas na Seção 3.3.

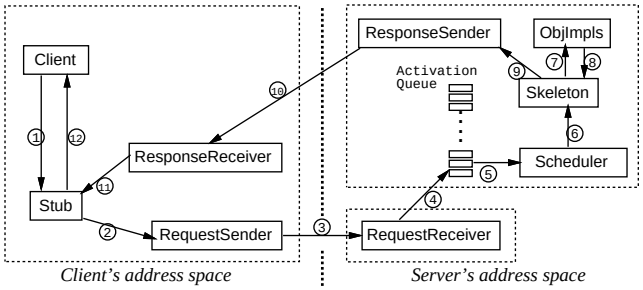

Figura 4: Entidades envolvidas no processo de comunicação entre *stubs* e *skeletons* em RME. A região pontilhada indica a área de execução de uma *thread* e os números indicam a ordem em que métodos são executados durante a invocação remota.

A utilização de quatro processadores de serviço facilita o processo de reconfiguração de RME. Por exemplo, caso necessário remover a fila de requisições apresentada na Figura 4, basta alterar a implementação das classes `RequestReceiver` e `RequestSender`. Uma vez que todos os processadores de serviço são criados via fábricas de objetos, tal modificação é simples, não tendo impacto sobre outros componentes do sistema.

3.3 A Estrutura do Servidor

Chamadas remotas são tratadas por um servidor de requisições cuja estrutura foi parcialmente descrita na Seção 3.1. A estrutura de tais servidores segue um padrão de projeto conhecido como *active object* [18]. Este padrão separa o processamento necessário à invocação de um método do processamento necessário à sua execução a fim de simplificar o acesso a um objeto que reside em sua própria *thread* de controle. Na Figura 5 estão representados os principais componentes deste padrão.

De acordo com o padrão *active object*, invocações de métodos não são feitas diretamente sobre o objeto responsável por executá-las. Ao invés disto, tais chamadas são inseridas em uma fila de prioridade. A fim de representar uma chamada remota, foi definida uma classe denominada `MethodRequest`. A fila de prioridade recebe instâncias desta classe, que contêm informações tais como os argumentos da chamada, um identificador que especifica o objeto remoto ao qual tal chamada se refere e a prioridade da requisição. Em RME, a fila de prioridade utiliza um heap [4] para efetuar as inserções e remoções de elementos, a fim de permitir que tais operações sejam realizadas de forma eficiente.

Assim como a plataforma CORBA [7], o sistema RME utiliza uma semântica conhecida como *at most once* para caracterizar o processamento de chamadas

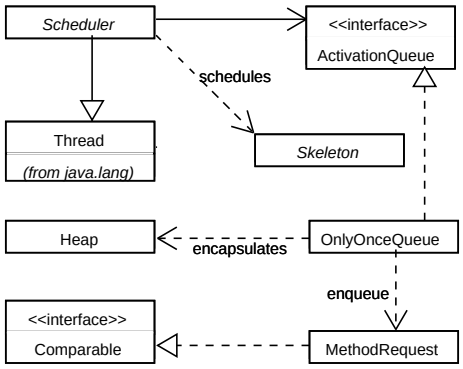

Figura 5: Principais classes que compõem um servidor de chamadas remotas

remotas. Isto quer dizer que, dada uma invocação remota de método, esta é executada no máximo uma vez, podendo, eventualmente, não ser processada. A fim de garantir esta semântica, para cada requisição remota é criado um identificador único, o qual é armazenado na fila de prioridade. Requisições que apresentem identificadores já encontrados são descartadas.

Conforme descrito na Seção 3.1, as requisições armazenadas na fila de prioridade são processadas por uma entidade conhecida como escalonador. Cada servidor remoto, em RME, possui somente um escalonador, embora este possa utilizar diversas *threads* para o processamento de chamadas. Políticas comuns são a criação de uma *thread* para cada requisição que chega à fila de prioridade ou o uso de uma única *thread* para processar todas as chamadas remotas, sendo esta última alternativa a correntemente utilizada em RME, embora ambas tenham sido implementadas.

Um escalonador possui uma coleção de referências para *skeletons*, os quais se encarregam de passar para os objetos remotos as requisições recebidas. *Skeletons* podem ser conectados ao escalonador dinamicamente. Entretanto, caso uma requisição seja retirada da fila em um momento em que o escalonador não possua referência para um *skeleton* capaz de tratá-la, a mesma é descartada. Nesse caso, o escalonador se encarrega de transmitir uma exceção para o endereço de retorno da requisição descartada. A fim de garantir que cada requisição seja tratada pelo objeto ao qual se destina, elas são acrescidas com o identificador de tal objeto, o qual é único em uma aplicação distribuída.

3.4 *Caches* Persistentes e *Flyweitgh* Stubs

Para diminuir o número de conexões e melhorar o aproveitamento da rede sem fio, *stubs* podem ser acrescidos com um *cache* que associa os argumentos de uma chamada remota ao valor retornado pela mesma. Com este fim, o usuário de RME pode definir um decorador de *stubs* que intercepta as invocações, armazenando seus parâmetros e o correspondente valor produzido em uma

tabela. Antes de enviar qualquer solicitação remota, esta tabela é verificada e, caso a chamada em questão já tenha sido processada, o resultado da execução anterior é retornado sem que qualquer troca de mensagens aconteça. Ao definir decoradores deste tipo, o usuário deve somente interceptar chamadas que não originem efeitos colaterais, isto é, que não causem modificações no estado do objeto remoto e que sempre retornem valores iguais para os mesmos argumentos passados.

Restrições quanto ao consumo de energia podem levar à interrupção de uma aplicação baseada na plataforma RME. Neste caso, é possível que os valores armazenados em *caches* de *stubs* sejam salvos em memória persistente. Devido a este fato, as fábricas de *stubs*, antes de criar tais objetos, sempre procuram pela existência de arquivos contendo dados a fim de compor *caches*. Caso tais arquivos existam, as novas instâncias de *stubs* são criadas com os dados encontrados.

A pequena quantidade de memória disponível nos dispositivos para os quais se destina a plataforma RME pode comprometer a utilização dos *caches* anteriormente descritos. A fim de contornar este problema, RME utiliza o padrão de projeto conhecido como *flyweight* [6] a fim de reduzir o número de instâncias diferentes de *stubs* em uma aplicação distribuída. Este padrão procura fatorar as informações comuns a diversos objetos em um único componente. Os demais dados, que tornam tais objetos únicos, passam a ser utilizados como parâmetros de métodos. Em RME este padrão é utilizado de modo a assegurar que em uma aplicação exista somente uma instância de *stub* para um dado objeto remoto. Isto quer dizer que, se em uma aplicação distribuída existem diversas referências remotas para o mesmo objeto, todas elas compartilham um *stub* comum.

Tanto a utilização de *caches* em *stubs* e a possibilidade de armazená-los de forma persistente quanto a utilização do padrão *flyweight* visando a economia de recursos são mecanismos opcionais em RME. Quaisquer destas técnicas podem ser usadas de forma independente ou em conjunto, desde que a fábrica de *stubs* apropriada seja utilizada pela aplicação.

3.5 Objetos Remotos

Em RME, objetos remotos, isto é, objetos cujos métodos poderão ser invocados remotamente, devem estender a classe `RemoteObject`. Tal classe é responsável por definir a semântica de algumas operações comuns a objetos na linguagem Java, como `equals`, `hashcode` e `toString`. `RemoteObject` implementa uma interface que define dois métodos responsáveis pela forma como um servidor de métodos remotos deve ser inicializado e finalizado. São estes os métodos `activateObject` e `deactivateObject`. Na classe `RemoteObject`, tais métodos foram declarados abstratos a fim de aumentar

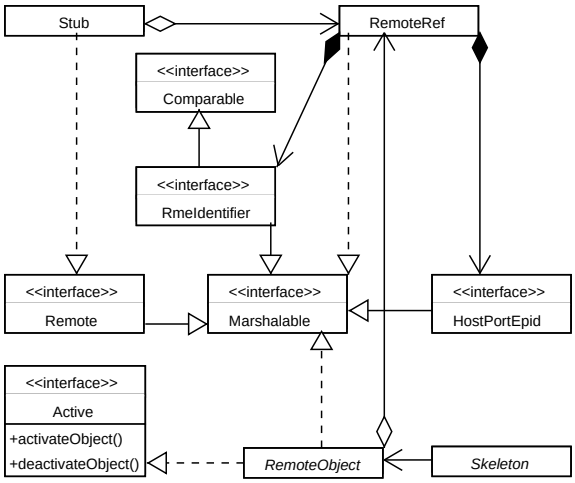

Figura 6: Iteração entre objeto remoto, referência remota, *stub* e *skeleton*

as possibilidades de reconfiguração de RME. O método `activateObject` é responsável por criar um *skeleton* para o objeto remoto e registrar este *skeleton* em um escalonador. Além disso, `activateObject` inicializa um receptor de conexões para receber as requisições destinadas ao objeto remoto. O método `deactivateObject`, por sua vez, tem a função de remover do escalonador o *skeleton* correspondente ao objeto remoto em questão e interromper o receptor de conexões anteriormente criado.

A semântica de referências remotas é definida em RME pela classe `RemoteRef`. Objetos desse tipo permitem que *stubs*, localizados no espaço de endereçamento de aplicações clientes, possam fazer referência aos objetos remotos que representam. Uma referência remota contém duas informações: um identificador que caracteriza de modo único o objeto remoto que lhe corresponde e um endereço dado pelo par (*endereço IP*, *número de porta*). Tais informações são, respectivamente, representadas pelas classes `RmeIdentifier` e `HostPortEpid`.

A Figura 6 mostra as iterações existentes entre as principais classes descritas nesta seção. A interface `Marshalable` caracteriza, em RME, objetos que podem ser transformados em seqüências de *bytes* e enviados via um canal de comunicação. Esta interface será descrita na Seção 3.6.

3.6 Passagem de Objetos por Valor

Conforme descrito na Seção 1, em J2ME objetos não podem ser serializados segundo os mesmos princípios utilizados nas outras versões da linguagem Java, devido à ausência, nessa plataforma, dos mecanismos de reflexividade encontrados naquelas edições. A fim de contornar esse problema e permitir que objetos sejam passados como parâmetros ou como valores de retorno de chamadas remotas, sem, contudo, comprometer ca-

racterísticas da linguagem como polimorfismo e sobrecarga, em RME foi definida uma interface denominada `Marshalable`. Esta interface contém dois métodos: `marshal` e `unmarshal`, que definem, respectivamente, como um objeto pode ser transformado em uma seqüência de *bytes* e como pode ser recuperado a partir de uma seqüência desse tipo. Em RME, apenas objetos que implementam essa interface podem ser enviados por um canal de comunicação.

A solução adotada em RME transfere para o desenvolvedor de aplicações a tarefa de serializar os objetos, porém define um protocolo de serialização. Com este fim, em RME foi implementada a classe `RmeStream`, a qual contém rotinas para converter em *bytes* tipos primitivos como inteiros e valores booleanos, além de tipos compostos como arranjos de objetos e *Strings*. Para a serialização de objetos, `RmeStream` implementa os métodos `readObject()` e `write(Marshalable)`, os quais fazem uso dos métodos `marshal` e `unmarshal` implementados pelo usuário.

O método `write` utiliza o método `marshal` do objeto a ser serializado para gerar uma seqüência de *bytes* que o representa e anota tal cadeia com o nome da classe a que pertence o objeto. O processo inverso, de criação de um objeto a partir da seqüência de *bytes* que o descreve, faz uso desse nome para obter uma instância da classe nomeada. Para preencher os atributos da instância assim obtida, o método `unmarshal` da mesma é então aplicado à cadeia formada pelos *bytes* remanescentes.

Objetos que descrevem exceções também podem ser serializados segundo o protocolo de serialização definido em RME desde que implementem a interface `Marshalable`. Isto é necessário porque caso um método remoto ative uma exceção enquanto estiver sendo processado, o objeto cliente precisa ser notificado da ocorrência do erro. A fim de permitir esta notificação, exceções levantadas por um método remoto sempre são passadas de volta para o cliente no lugar do valor de retorno esperado. A extração de um objeto que denota uma exceção força o seu disparo, o que permite que erros gerados em espaços de endereçamento remotos interfiram no fluxo de execução de aplicações clientes.

3.7 O Protocolo de Comunicação RMEP

O protocolo JRMP [9], utilizado pela plataforma Java RMI para o transporte de dados, é pouco adequado ao ambiente no qual executam computadores móveis [3]. 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 destas, somente duas mensagens conteriam informação pertinente à invocação remota propriamente dita [3]. 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 o menor número possível de mensagens para permitir a comunicação entre objetos distribuídos. A fim de reduzir o número de mensagens necessárias a uma chamada remota, RMEP não utiliza mensagens do tipo `Ping` para verificar se conexões existentes ainda podem ser utilizadas, como é feito em Java RMI [3]. Em RME novas conexões são criadas sempre que uma invocação remota precise ser executada e são destruídas ao término da mesma. Este padrão de criação de conexões é dispendioso, pois invocações remotas do mesmo método sobre o mesmo servidor originam o estabelecimento de novos canais, sem que recursos já criados sejam reaproveitados. Embora tal abordagem possa parecer inadequada, é preciso lembrar que aplicações em computadores móveis tendem a ser executadas durante curtos períodos de tempo, e normalmente apenas para pequenas consultas, de modo que poucas conexões tendem a existir simultaneamente em máquinas clientes.

O protocolo RMEP utiliza quatro tipos de mensagens diferentes, as quais foram nomeadas *call*, *return*, *ping* e *ack*. Mensagens do tipo *call* caracterizam as invocações remotas de métodos e são respondidas por mensagens do tipo *return*, as quais contêm valores que resultam da execução de algum método remoto. As mensagens do tipo *ping* e *ack* são utilizadas para verificar se existem servidores de métodos remotos ativos em um dado endereço.

A Tabela 1 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 `Rme-`

$message$	$\rightarrow$	$header\ version\ msg_{type}$
$header$	$\rightarrow$	`0x52 0x4d 0x45 0x50`
$version$	$\rightarrow$	`0x01`
msg_{type}	$\rightarrow$	$msg_{req} \mid msg_{ack}$
msg_{req}	$\rightarrow$	`epid` req_{type}
req_{type}	$\rightarrow$	`call` $call_{id}\ obj_{id}\ op\ args$
	$\mid$	`ping`
$call_{id}$	$\rightarrow$	`id`
obj_{id}	$\rightarrow$	`id`
op	$\rightarrow$	`int`
$args$	$\rightarrow$	`byteStream`
msg_{ack}	$\rightarrow$	`return` $call_{id}\ ret_{val}$
	$\mid$	`ack`
ret_{val}	$\rightarrow$	`byteStream`

Tabela 1: Formato de mensagens do protocolo RMEP

`Identifier`, representada, na Tabela 1 pelo símbolo `id`. Finalmente, os argumentos de uma chamada remota e o valor de retorno da mesma são representados pelo símbolo `byteStream`, que denota uma seqüência de *bytes* obtida da serialização de objetos e de tipos primitivos.

3.8 O Serviço de Localização de Nomes

A fim de que o serviço de invocação remota de métodos possa ser plenamente utilizado, é necessário disponibilizar à aplicação uma maneira de localizar objetos distribuídos. Em Java RMI, tal serviço de localização está disponível por meio de uma classe denominada `Naming` [9]. Na plataforma RME foi definida a classe `RmeNaming`, que disponibiliza ao usuário as mesmas funcionalidades encontradas em Java RMI, tais como a associação de nomes a objetos remotos 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.

O servidor de nomes utiliza uma *thread* separada para verificar quais referências nele armazenadas permanecem válidas. Esta *thread* envia periodicamente mensagens do tipo `ping`, descritas na Seção 3.7, para os objetos que cadastraram referências remotas no servidor de nomes. Os objetos que não responderem às mensagens enviadas em um certo intervalo de tempo têm suas referências removidas da tabela de nomes. Isto evita que referências para componentes que não mais existam sejam enviadas para objetos clientes.

3.9 A API de RME e um Exemplo de Uso

Aplicações distribuídas que utilizam o pacote RME são sintaticamente semelhantes àquelas desenvolvidas a partir de Java RMI. Assim como em Java RMI, uma interface é utilizada para definir um conjunto de métodos que poderão ser invocados remotamente. Em ambas as plataformas, estas interfaces devem estender uma outra interface, denominada `Remote`, que as caracteriza como passíveis de serem utilizadas remotamente. A Figura 7 exemplifica, em RME, uma interface onde é declarado um método remoto. Esta interface caracteriza um servidor que responde a consultas de números telefônicos. Tal servidor recebe pesquisas contendo nomes de pessoas e retorna uma cadeia de caracteres representando os números telefônicos associados aos nomes recebidos.

```
import arcademis.*;
public interface PhoneBook extends Remote {
    public String getPhone(String name)
    throws ArcademisException;
}
```

Figura 7: Definição de uma interface remota em RME

Observe que o pacote importado pelo programa mostrado na Figura 7 é denominado `arcademis`, e não `rme`. Tal fato se justifica porque a plataforma RME foi implementada como uma instância de um arcabouço para desenvolvimento de *middlewares* orientados por objetos denominado *arcademis*. Maiores detalhes sobre esse arcabouço podem ser obtidos em [12].

Objetos remotos, ou seja, as entidades que devem implementar os métodos de interfaces remotas, como a mostrada na Figura 7, precisam estender a classe `RemoteObject`. A implementação destes objetos, contudo, não precisa levar em consideração o fato de seus métodos poderem ser utilizados remotamente. A Figura 8 mostra uma implementação que poderia ser provida para a interface apresentada na Figura 7. No exemplo mostrado, `PhoneAddress` é uma classe utilizada para associar ao nome de uma pessoa suas informações particulares como número telefônico e endereço domiciliar. É importante ressaltar que todas as inicializações relacionadas à semântica distribuída do objeto ficam a cargo do método `activateObject()`, que pertence à classe `RemoteObject` e também do método construtor desta mesma classe. O método construtor, por exemplo, se encarrega de criar um *skeleton* e uma referência remota para o objeto.

Para que um objeto remoto como o mostrado na Figura 8 possa ser utilizado por aplicações distribuídas, é necessário que ele seja registrado em um servidor de nomes e ativado. Um exemplo de aplicação em que tais procedimentos são executados é mostrado na Figura 9. O método `activateObject()` inicializa um servidor responsável por receber requisições remotas e associa o *skeleton* criado pelo método construtor de `RemoteObject` a um escalonador, o qual é responsável por definir a ordem em que invocações são repassadas à implementação do objeto remoto, conforme descrito

```
import rme.*;
import arcademis.*;
public class PhoneBook_Impl extends RemoteObject
implements PhoneBook {
    Hashtable h = null;
    public String getPhone(String name) {
        PhoneAddress addr = (PhoneAddress)h.get(name);
        if(addr == null) return null;
        else return addr.getPhoneNumber();
    }
    public void putAddress(PhoneAddress addr) {
        h.put(addr.getName(), addr);
    }
}
```

Figura 8: Implementação de uma interface remota

```
import rme.*;
import arcademis.*;
public class PhoneBookServer {
  public static void main(String[] args) {
    RmeConfigurator c = new RmeConfigurator();
    c.configure();
    try {
      PhoneBook_Impl phoneList = new PhoneBook_Impl();
      RmeNaming.bind("phone_book", phoneList);
      phoneList.activateObject();
      // put the code to fill the list with phone
      // information here
    } catch (Exception e) {}
  }
}
```

Figura 9: Inicialização de um objeto remoto

na Seção 3.3.

Uma aplicação precisa definir a configuração da plataforma RME que será utilizada a fim de poder realizar chamadas remotas. Estas definições incluem a escolha das fábricas de objetos que irão compor a camada RME da aplicação. Para este fim, o pacote disponibiliza ao desenvolvedor a classe `RmeConfigurator`, a qual contém algumas opções de configuração que garantem o funcionamento do sistema. Caso seja necessário alterar componentes da plataforma RME, uma extensão da classe configuradora pode ser utilizada. Para a localização de objetos distribuídos, por sua vez, a classe `Naming` implementa o método `lookup`, que retorna, a partir de um nome de objeto, um *stub* para o mesmo. Um exemplo de aplicação para a plataforma J2ME que utiliza o sistema RME é mostrado na Figura 10.

3.10 Análise de Desempenho

Para medir o desempenho da plataforma RME foram feitos alguns testes utilizando como cliente um emulador de telefone celular cuja máquina virtual K (KVM), executa 100 *bytecodes* por milisegundo. Tanto o servidor de requisições quanto o emulador de celular foram executados em computadores de igual capacidade: *Pentium* 4 de 2.0GHz e 512MB de memória disponível conectados por uma rede *ethernet* de 10Mb/s. Foi me-

```
import rme.*;
import arcademis.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SimpleClient extends MIDlet
implements CommandListener {
  private PhoneBook phoneList = null;
  private Command enterCommand;
  private TextBox tb;
  public SimpleClient() {
    RmeConfigurator mc = new RmeConfigurator();
    mc.configure();
    try {
      phoneList = (PhoneBook)
      RmeNaming.lookup("algol.dcc.ufmg.br\\phone_book");
    } catch (Exception e) {}
    enterCommand = new Command("Enter", Command.OK, 1);
    tb = new TextBox("Example","Send",15,TextField.ANY);
    tb.addCommand(enterCommand);
    tb.setCommandListener(this);
  }
  protected void startApp() {
    Display.getDisplay(this).setCurrent(tb);
  }
  protected void pauseApp() {}
  protected void destroyApp(boolean u) {}
  public void commandAction(Command c, Displayable d) {
  if (c == enterCommand) {
    try {
      tb.setString(phoneList.getPhone("Lima Barreto"));
    } catch(ArcademisException e){}
  }
}
```

Figura 10: Aplicação cliente implementada em J2ME

dido o número de requisições tratadas por segundo para seis métodos que não possuem argumentos e cujos valores de retorno são diferentes. Na Tabela 2, para cada método, é mostrado o valor médio apurado a partir da execução de 10 séries de 50 invocações remotas.

A fim de fornecer um limite superior para o maior número de requisições que podem ser tratadas no cenário anteriormente descrito, 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. Essa aplicação utiliza a mesma implementação de canal utilizada 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. Além disso, assim como em RME, também nessa aplicação uma nova conexão é criada para simular cada invocação remota. Tanto a aplicação baseada em `sockets` quanto aquela baseada em RME são constituídas por um cliente implementado em J2ME e por um servidor implementado em J2SE. Os resultados apurados são mostrados na Tabela 2.

Pela análise da Tabela 2, nota-se que quanto mais complexo o valor de retorno de uma chamada remota, maior o tempo necessário para completá-la. Tal atraso

Valor de retorno	tamanho da resposta (*bytes*)	req./s RME	req./s `sockets`
`byte`	36	4.39	5.00
`short`	37	4.38	5.00
`int`	39	4.49	4.98
`long`	43	4.35	4.99
`String`	48	3.89	4.99
`String[]`	168	2.51	4.98

Tabela 2: Testes de desempenho realizados sobre uma aplicação RME

deve-se ao algoritmo de serialização utilizado na implementação de RME. O tempo de execução desse algoritmo é relevante dadas as características da máquina cliente: um emulador capaz de executar 100 *bytecodes* por milisegundo. É possível observar também que o tamanho das mensagens transmitidas não é um fator decisivo para o desempenho da aplicação. Esse fato se deve à utilização de um *buffer* na implementação do canal de transmissão, pois tal estrutura permite que todos os *bytes* que constituem uma mensagem sejam enviados em um mesmo pacote de dados.

4 Trabalhos Relacionados

Existem diversas plataformas *middleware* que utilizam o modelo de programação orientada por objetos no desenvolvimento de aplicações distribuídas. Um dos primeiros exemplos desse tipo de sistema foi implementado na linguagem Modula 3, no início da década de 90 [2]. Vários dos conceitos e algoritmos empregados nessa implementação pioneira, como o mecanismo de coleta de lixo distribuída e as técnicas de serialização de objetos são atualmente utilizados em muitas das plataformas de *middleware* mais populares, como Java RMI, por exemplo.

Plataformas de *middleware* como CORBA [7], .NET [5] ou Java RMI são sistemas complexos e monolíticos e, devido a este fato, sua utilização em ambientes caracterizados pela limitação de recursos fica comprometida. Algumas tentativas de portar tais plataformas para o ambiente móvel foram realizadas, principalmente com relação ao sistema CORBA. Neste contexto, um conjunto mínimo da especificação CORBA foi definido justamente com o intuito de permitir o desenvolvimento de aplicações compatíveis com aquela plataforma e que utilizassem somente os recursos estritamente necessários. Tal subconjunto de CORBA é conhecido como *minimumCORBA* [8].

Existem diversos sistemas de *middleware* que, assim, como RME, permitem reconfiguração estática, como, por exemplo, a plataforma *TAO* [14], em cuja implementação foram propostos diversos padrões de projeto incorporados na implementação de RME [16]. Este sistema utiliza arquivos contendo descrições de configurações para especificar as estratégias que serão utilizadas, por exemplo, para tratamento de conexões, concorrência e escalonamento, as quais são definidas estaticamente durante a inicialização das aplicações. A principal característica de tais sistemas é o fato de permitirem a implementação de plataformas de *middleware* contendo apenas as funcionalidades necessárias para as aplicações a que se destinam. Por exemplo, é possível configurar uma instância de RME contendo somente o código necessário à parte cliente de uma aplicação, ou somente o código necessário à parte servidora da mesma.

O sistema conhecido como *UIC CORBA* [14], assim como RME, foi desenvolvido com o objetivo de prover alternativas adequadas ao desenvolvimento de aplicações distribuídas para computação móvel. UIC permite que desenvolvedores de aplicações utilizem opções de reconfiguração estáticas e dinâmicas, isto é, permite que o sistema seja alterado durante a sua execução. Para esse fim, o sistema define um arcabouço de componentes abstratos que encapsulam os aspectos considerados comuns a toda plataforma compatível com CORBA, como os protocolos de transporte e serialização utilizados. Classes concretas, carregadas dinamicamente, implementam essas propriedades de acordo com as necessidades de uma determinada plataforma de *middleware*.

A plataforma *LegORB* [15] é outro sistema de *middleware* compatível com CORBA e que disponibiliza aos desenvolvedores de aplicações grande número de opções de reconfiguração. O sistema permite tanto modelos de configuração estáticos quanto dinâmicos. A versão estática fornece os melhores resultados em termos de utilização de memória. A título de exemplo, plataformas obtidas via configuração estática não ocupam mais que 23KB de memória, ao passo que sistemas dinamicamente configurados ocupam em torno de 140KB de memória.

Os sistemas anteriormente apresentados foram implementados em C++ e são compatíveis com a plataforma CORBA, ao passo que RME é completamente implementado na linguagem Java. Por outro lado, existe uma implementação de RMI para a configuração CDC, apresentada na Seção 2, que está disponível como um pacote opcional de J2ME. Este sistema, conhecido como *RMI OP* (*RMI Optional Package*) [10], permite que aplicações clientes em dispositivos CDC manipulem objetos localizados em servidores remotos. A principal diferença entre RMI OP e RME são as configurações para as quais tais sistemas se destinam. Enquanto RME foi desenvolvido para CLDC, o pacote RMI OP não é suportado por esta configuração.

Os sistemas LegORB, UIC CORBA e RME têm o objetivo de facilitar o desenvolvimento de aplicações distribuídas para computação móvel. Sendo assim, a

RME	34.5KB
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 3: Comparação entre o tamanho das bibliotecas dos sistemas LegORB, UIC CORBA e RME

utilização eficiente da memória disponível é um requisito essencial que deve ser atendido por essas plataformas. A Tabela 3 mostra o tamanho das bibliotecas que compõem cada um destes sistemas. Essa tabela trata somente das bibliotecas necessárias à parte cliente de uma aplicação distribuída, sendo que, no caso se RME, o espaço ocupado pela máquina virtual Java não está sendo levado em consideração. Caso seja necessário, existem algumas maneiras de reduzir o tamanho do pacote RME. Por exemplo, conforme discutido na Seção 3, a implementação de RME utiliza fábricas para a criação de objetos. Embora tais fábricas facilitem a reconfiguração do sistema, elas suas bibliotecas. A fim de reduzir esse tamanho, as fábricas podem ser removidas e os objetos podem ser instanciados diretamente.

5 Conclusões e Trabalhos Futuros

Este trabalho apresentou o *middleware* RME, que foi desenvolvido com o intuito de prover um serviço de chamada remota de métodos para o perfil MIDP/CLDC da plataforma J2ME. Essa é a principal contribuição da plataforma RME, uma vez que não existe ainda nenhum sistema desse tipo para CLDC cuja implementação esteja consolidada. Além de descrever os principais detalhes relativos à implementação da plataforma RME, como por exemplo os padrões de projeto utilizados e o protocolo de transmissão de dados, foi descrito um exemplo de aplicação distribuída baseada nos serviços fornecidos pelo sistema e também foram mostradas comparações entre RME e alguns dos sistemas semelhantes mais conhecidos.

A implementação de RME utilizou diversos padrões de projeto, o que deu origem a um sistema bastante reconfigurável. Enquanto os sistemas de *middleware* mais populares possuem um projeto monolítico, RME é constituído por um conjunto de componentes que implementam funcionalidades básicas e que podem ser removidos ou alterados de maneira simples. Dessa forma, o sistema RME pode ser utilizado para definir aplicações clientes, servidoras ou que agregam ambas capacidades, podendo enviar e receber invocações remotas de métodos. Essa propriedade é útil no desenvolvimento de aplicações para telefonia celular porque em muitos casos não há necessidade dessas atuarem como servidores de requisições remotas.

O pacote RME desenvolvido para a plataforma J2ME não é compatível com a versão J2SE da linguagem Java. Basta, contudo, alterar duas classes para portar o sistema de uma plataforma para a outra. Essas classes, denominadas `TcpSocketChannel` e `TcpSocketServer`, utilizam os mecanismos particulares que cada plataforma Java provê para o estabelecimento de conexões e tais funcionalidades não são as mesmas nos dois ambientes. Dada a utilização de fábricas para a instanciação de objetos essa modificação é trivial. Assim, com o intuito de validar o sistema foram implementados servidores de métodos remotos em J2SE e clientes baseados em J2ME.

As bibliotecas utilizadas por aplicações clientes, as quais incluem o serviço de localização de nomes, os protocolos de transmissão e serialização, as definições de referências remotas e *stubs*, todas as classes necessárias ao estabelecimento e manutenção de conexões, bem como ao envio de dados, além de uma série de funcionalidades para tratamento de erros ocupam cerca de 34.5KB de memória. As bibliotecas utilizadas por aplicações servidoras de chamadas remotas, por outro lado, ocupam cerca de 67.9KB. Tais bibliotecas, além de conterem todas as classes que dão suporte às aplicações clientes, contêm também servidores de conexões, um gerenciador de evento, a fila de prioridade na qual são armazenadas requisições remotas, definições de objetos remotos e *skeletons*, além de um escalonador que determina a ordem em que requisições serão atendidas.

Dentre as características da plataforma RME que a tornam adequadas para a computação móvel, citam-se, em primeiro lugar, o caráter modular do sistema, o qual permite a remoção ou adição de componentes de acordo com as necessidades da aplicação. Além disto, RME utiliza um protocolo otimizado, tanto com relação ao tamanho das mensagens trocadas quanto com relação à quantidade delas, o que se mostra apropriado à pequena banda de transmissão que dispositivos móveis têm disponível. Do caráter reconfigurável da plataforma, advêm outras características que a tornam propensa a ser utilizada no ambiente móvel. Desse modo, conforme descrito na Seção 3.4, pode-se utilizar o padrão *flyweight* para limitar o número de instâncias de *stubs* existentes em uma aplicação, quando restrições de memória assim o exigirem. E pode-se, ainda, utilizar *caches* em *stubs* para diminuir o tráfego de mensagens na rede.

Como trabalhos futuros, espera-se prover uma configuração de RME capaz de efetuar chamadas assíncronas de métodos. Esta capacidade é particularmente interessante no contexto da computação móvel, pois, dadas as restrições de energia dos dispositivos utilizados, em muitos casos tais aparelhos não são capazes de se manterem conectados durante todo o tempo de

processamento da chamada síncrona. Uma vez que o modelo RME já permite o estabelecimento de conexões assíncronas, acredita-se que a extensão desse mecanismo para permitir o assincronismo de chamadas remotas será simples. Conexões ditas assíncronas não bloqueiam a execução da aplicação caso não possam ser efetivadas de imediato. Nesse caso, novas tentativas de estabelecimento de uma conexão seguem a tentativa inicial até que sua efetivação seja possível.

Além de introduzir chamadas assíncronas ao sistema, espera-se acrescentar algumas possibilidades de reconfiguração dinâmica à plataforma RME. Alguns estudos nesse sentido já foram feitos, como permitir, por exemplo, a adoção de algoritmos de criptografia ou compactação que seriam aplicados às mensagens trocadas sem interromper o funcionamento do sistema. Com esse fim o protocolo de comunicação deveria ser alterado de modo a possibilitar que as partes interessadas pudessem negociar a adoção de tais algoritmos.

Finalmente, pretende-se fornecer para RME a implementação do protocolo JRMP, utilizado por Java RMI. Tal adição permitirá que aplicações desenvolvidas nas duas plataformas possam interagir diretamente. Atualmente, tal iteração somente é possível via classes adaptadoras, as quais se encarregam de permitir a comunicação entre os protocolos RMEP e JRMP. Essa abordagem, todavia, não é prática, pois para cada interface remota é necessário implementar um adaptador.

Código fonte O pacote RME está disponível para cópia sem restrições no endereço eletrônico: `http://www.-dcc.ufmg.br/~fernandm/arcademis/`.

Agradecimentos Esta pesquisa foi financiada pela Fapemig e pelo CNPq.

Referências

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

[2] 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.

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

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

[5] Microsoft Corporation. Microsoft .NET Development. msdn.microsoft.com/net/ – última visita: junho de 2003.

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

[7] 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.

[8] Object Management Group. Minimumcorba specification, 2002. doc.ece.uci.edu/CORBA/formal/02-08-01.pdf – última visita: junho de 2003.

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

[10] Sun Microsystens Inc. Rmi optional package specification version 1.0. http://java.sun.com/products/rmiop/ – última visita: junho de 2003.

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

[12] Fernando Magno Quintão Pereira. Arcademis's home page. http://www.dcc.ufmg.br/∼fernandm/arcademis – última visita: september 2003.

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

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

[15] 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.

[16] 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.

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

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

[19] 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, May 2000.

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

[21] 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.