UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

HMI: UM MIDDLEWARE PARA OBJETOS DISTRIBUÍDOS SOBRE O PROTOCOLO HTTP

ABEL LUIZ CECHINEL

BLUMENAU
2008

2008/2-01

ABEL LUIZ CECHINEL

HMI: UM MIDDLEWARE PARA OBJETOS DISTRIBUÍDOS SOBRE O PROTOCOLO HTTP

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Paulo Fernando da Silva , Mestre - Orientador

BLUMENAU
2008

2008/2-01

HMI: UM MIDDLEWARE PARA OBJETOS DISTRIBUÍDOS SOBRE O PROTOCOLO HTTP

Por

ABEL LUIZ CECHINEL

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Paulo Fernando da Silva, Mestre – Orientador, FURB
Membro:	Prof. Adilson Vahldick, Mestre – FURB
Membro:	Prof. Francisco Adell Péricas, Mestre – FURB

Blumenau, 10 de Fevereiro de 2009

Para Mara, Letícia e Vinícius.

AGRADECIMENTOS

Esta é a última página que escrevo deste trabalho. A mais prazerosa e a que está sendo mais demorada. Não por falta do que agradecer. Pelo contrário, tantas pessoas me ajudaram a chegar aqui, que é impossível escrevê-la sem cometer injustiça.

Mas há alguém que preciso agradecer primeiro: minha amada esposa Mara. Ela não deu nenhuma dica sobre este trabalho. Nem o leu ou corrigiu parágrafo algum. Mas tudo que está aqui devo a ela. Obrigado por ter me apoiado quando precisei parar de estudar e por ter me incentivado a retornar.

Este trabalho não seria possível sem o apoio do meu orientador e amigo, Paulo Fernando da Silva. Agradeço pela confiança depositada, principalmente porque, quando o procurei para propor o trabalho, nem me conhecia.

Ao amigo Alexander Roberto Valdameri, pela presteza e incentivo por ocasião do meu reingresso.

Agradeço a todos os professores pelas inestimáveis horas de atenção, que me adicionaram valiosos conhecimentos.

À Deus, pela vida.

...qualquer um que deixe para trás um manual escrito e da mesma forma qualquer um que o receba, acreditando que tal obra será clara e certa, deve ser extremamente ingênuo...

Platão

RESUMO

Este trabalho apresenta o desenvolvimento de um middleware para objetos distribuídos na web. Buscando compatibilidade com firewalls, o middleware utiliza o HTTP como protocolo de aplicação. O middleware desenvolvido tem um código na ordem de 30 KBytes, sendo suportado por dispositivos móveis e compatível com Java Micro Edition. São abordados no trabalho, a especificação e implementação dos componentes de uma arquitetura de objetos distribuídos, a interface remota, referência remota, módulo de comunicação e coletor de lixo distribuído. Por fim, é apresentado um estudo de caso com implementação de um sistema distribuído, utilizando o middleware para acesso a objetos remotos.

Palavras-chave: Objetos distribuídos. Middleware. JME. Dispositivos móveis.

ABSTRACT

This work presents the development of a middleware for distributed objects on the web. Seeking compatibility with firewalls, the middleware uses the HTTP as the application protocol. The middleware has developed a code around 30 KBytes, being supported by mobile devices and compatible with Java Micro Edition. This work addresses the specification and implementation of the components of an architecture of distributed objects such as the remote interface, remote reference, the communication module and the garbage collector distributed. Finally, we presented a case study with implementation of a distributed system, using middleware to access remote objects.

Key-words: Distributed objects. Middleware. JME. Mobile devices.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Desde o surgimento dos computadores, há demanda por compartilhamento de recursos. As primeiras necessidades surgiram em função do hardware caro. Vários terminais precisavam compartilhar a mesma unidade central de processamento, as mesmas impressoras e as mesmas unidades de gravação.

Até o início da década de 80, os computadores eram extremamente grandes e caros. Poucas organizações possuíam um computador. Muitas empresas, como as Têxteis de Blumenau, por exemplo, montavam empresas birôs de processamento de dados, onde um único computador realizava o processamento de várias empresas.

Esta realidade começou a mudar a partir da primeira metade dos anos 80. Primeiramente com o surgimento dos microcomputadores de grande capacidade. Inicialmente, surgiram os micros de 8 bits, extremamente rudimentares, com poucos kbytes de memória e de utilidade apenas para os aficionados. Mas logo apareceram as máquinas de 16, 32 e 64 bits e a memória passou a ser contada em megas e gigas. Segundo Tanenbaum e Steen (2008, p. 1), os atuais computadores de mil dólares apresentam uma capacidade de processamento 10^{13} vezes maior que os primeiros computadores, que custavam dez milhões de dólares.

Outro avanço significativo foi a disseminação das redes de computadores, ocorrida a partir da década de 90. Segundo Tanenbaum e Steen (2008, p. 1), uma *Local Area Network* (LAN), permite que centenas de máquinas localizadas dentro de um edifício sejam conectadas de modo tal que as informações possam ser movimentas entre máquinas a taxas de 100 milhões a 10 bilhões de bits/s. Uma *Wide Area Network* (WAN), permite que milhões de máquinas no mundo inteiro sejam conectadas a velocidades que variam de 64 Kbits/s a gigabits por segundo.

Unindo-se a enorme capacidade de processamento dos microcomputadores modernos à capacidade de compartilhamento oferecida pelas redes, passou a ser possível a montagem de sistemas compostos por diversos computadores conectados. Segundo Tanenbaum e Steen (2008, p. 1), estes sistemas costumam ser denominados sistemas distribuídos, em comparação com os Sistemas Centralizados anteriores, que consistem em um único computador, seus periféricos e talvez, alguns terminais remotos.

Na última década, o crescimento da Internet popularizou o conceito de sistemas distribuídos. O usuário acessa parte da aplicação em seu *browser*, sem ter a mínima idéia de onde está rodando o restante do sistema que efetivamente processará suas requisições.

Os sistemas distribuídos, em sua maioria, rodam em ambientes heterogêneos, formados por computadores com diversas arquiteturas e sistemas operacionais. Em computação, quando há necessidade de integrar componentes com estas características, utiliza-se o conceito de *middleware*, uma camada adicional de software que provê uma interface comum à componentes heterogêneos (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 29).

Existem diferentes implementações de *middlewares* para integração de sistemas distribuídos. As primeiras plataformas utilizavam técnicas procedurais, com chamada remota de procedimento, do inglês *Remote Procedure Call* (RPC), e baseavam-se no modelo cliente-servidor. Neste contexto, uma aplicação cliente invoca um procedimento remoto, passando como parâmetros, todas as informações necessárias para que o processo servidor possa tratá-lo. Posteriormente, surgiram *middlewares* orientados a objetos, como *Common Object Request Broker Architeture* (CORBA) e Java RMI. Todos os modelos de *middleware* têm como princípio básico encapsular os detalhes de comunicação exigidos pelos diferentes sistemas operacionais e arquiteturas de rede, facilitando o desenvolvimento das aplicações distribuídas.

A categoria de *middlewares* orientada a objetos, foco deste trabalho, também é conhecida como Objetos Distribuídos. A vantagem da aplicação de técnicas de objetos distribuídos é que leva-se para os sistemas distribuídos o paradigma da Programação Orientada a Objetos (POO). Neste contexto, além de abstrair detalhes da transmissão de dados entre os computadores da rede, o *middleware* de objeto distribuído, deve prover todas as características da POO como herança e polimorfismo, por exemplo.

Atualmente, alternativas "não orientadas a objetos" têm surgido com força no mercado, como *Web Services* e *Service Oriented Architecture* (SOA). Embora ofereçam facilidades por utilizarem o *HyperText Trasnfer Protocol* (HTTP) como protocolo de aplicação, estas alternativas provocam uma "mistura" de paradigmas nas aplicações desenvolvidas com técnicas de POO. Toda aplicação fica orientada à objetos, exceto as chamadas remotas. JAVA REMOTE METHOD INVOCATION (2008), diz que no baixo nível, os *middlewares* orientados a objetos fazem RPC, mas camadas de abstração adicionais, oferecem significativas vantagens aos programadores das linguagens orientadas a objetos.

A recente popularização da computação móvel tem aumentado significativamente a demanda por sistema distribuídos. Porém, as opções de *middlewares* disponíveis não são adequadas aos dispositivos portáteis, que em sua maioria têm pouca memória e limitada capacidade de processamento. Além disso, os *Web Services* exigem significativas configurações nos servidores e Java RMI e CORBA, abertura de portas de comunicação

específicas nem sempre compatíveis com as políticas, cada vez mais rígidas, de segurança de rede das empresas.

Para Herity (2006, tradução nossa), "*middlewares* para objetos distribuídos têm alcançado sucesso no processamento distribuído tradicional. CORBA, COM e Java RMI são implementações bem conhecidas do paradigma. Mas estas tecnologias são projetadas para computadores com alta capacidade de processamento, muita memória e sistemas operacionais com muitos recursos". Não há segundo ele, um *middleware* para objetos distribuídos adequado para sistemas embarcados.

Frente ao exposto acima, este trabalho se propõe a desenvolver um *middleware* com as seguintes características:

a) facilidade de implantação e o nível de abstração e aderência ao paradigma orientado a objetos semelhante do Java RMI;
b) facilidade de utilização na *Web* e compatibilidade com *firewalls* como os *Web servises*;
c) leve e configurável para ser suportado pelos dispositivos móveis.

Além das característica acima, o *middleware* deve ter a estratégia de serialização[1] dos objetos configurável. Isso é necessário porque o Java *Micro Edition* (JME) não provê serialização de objetos e neste caso, o programador da aplicação terá que implementar a serialização, seguindo uma especificação do *middleware*. Por outro lado, caso o *middleware* venha a ser utilizado num ambiente Java *Standard Edition* (JSE) ou Java *Enterprise Edition* (JEE), o que faz sentido pela facilidade com que lidará com *firewalls*, é desejável que utilize a serialização nativa do Java. Essa configuração do *middleware*, deverá ser via *eXtensible Markup Language (*XML), sem necessidade de alteração no código ou recompilações.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um *middleware para* objetos distribuídos sobre o HTTP que seja adequadamente suportado por dispositivos móveis.

Os objetivos específicos do *middleware* são:

a) fornecer um mecanismo para registro de serviços de objetos distribuídos no

[1]Técnica para transformar um objeto em uma seqüência de bytes para armazená-lo em arquivo ou enviá-lo pela rede (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 139).

servidor, tendo como única exigência de configuração a instalação de um *servlet*[2] *container*[3], como o *Tomcat*[4];

b) permitir que o cliente localize os serviços registrados no servidor;
c) tratar adequadamente os métodos de objetos distribuídos que retornam outro objeto distribuído. Neste caso, o retorno deve ser uma referência remota para o objeto e este deve ficar registrado até que seja liberado (item e) ou não mais referenciado (item f) pelo cliente;
d) suportar diferentes estratégias de serialização de objetos;
e) permitir que o cliente libere do servidor os objetos que não serão mais utilizados;
f) liberar do servidor os objetos que não tenham mais referência em nenhum cliente;
g) permitir que o programador gere *stubs*[5] e *skeletons*[6].

[2]Objeto Java capaz de receber uma requisição HTTP e gerar uma resposta com base na requisição recebida (FIELDS; KOLB, 2000, p. 7).

[3]*Web Server* que suporta a execução de *servlets.*

[4]*Servlet Container open source* do projeto Apache (APACHE TOMCAT, 2009).

[5]Padrão de projeto para objetos distribuídos que fornece ao programador do cliente uma classe concreta com a mesma interface pública do objeto distribuído registrado no servidor. A implementação dos métodos desta classe abstrai os detalhes da invocação remota (TANENBAUM; STEEN, 2008, p. 76).

[6]Padrão de projeto para objetos distribuídos que permite ao servidor passar requisições aos objetos registrados sem conhecer sua interface pública (TANENBAUM; STEEN, 2008, p. 268).

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo são apresentados as definições de sistemas distribuídos, objetos distribuídos e os *middlewares* para sua construção. Também são abordados os conceitos de computação móvel, o protocolo HTTP e suas particularidades na aplicação de sistemas distribuídos. Ao fina são apresentados dois trabalhos correlatos.

2.1 SISTEMAS DISTRIBUÍDOS

Segundo Coulouris, Dollimore e Kindberg (2007, p. 16), “definimos um sistema distribuído como sendo aquele no qual os componentes de hardware ou software, localizados em computadores interligados em rede, se comunicam e coordenam suas ações apenas enviando mensagens entre si”. Tanenbaum e Steen (2008, p. 1) apresentam sistemas distribuídos como “... um conjunto de computadores independentes que se apresenta a seus usuários com um sistema único e coerente”.

Para Tanenbaum (2007, p. 413), “os sistemas distribuídos são um tipo especial de sistemas com múltiplos processadores. São semelhantes aos multicomputadores no fato de cada nodo ter sua própria memória privada, com nenhuma memória física compartilhada. Contudo, os sistemas distribuídos são ainda mais fracamente acoplados do que os multicomputadores. Em contraste com os sistemas multicomputador que executam o mesmo sistema operacional, o mesmo sistema de arquivos e normalmente estão numa mesma sala ligados por uma rede de alta velocidade, um sistema distribuído, pode estar espalhado ao redor do mundo, executando em sistemas operacionais diferentes”.

Tanenbaum (2007, p. 413) aponta uma diferença importante entre sistemas multicomputadores e sistemas distribuídos, do ponto de vista das aplicações. Enquanto que o primeiro grupo é composto normalmente por grandes *racks* com equipamentos em uma sala-máquina, com o objetivo de resolver problemas computacionais intensivos, o segundo costuma ser composto por máquinas ligadas pela Internet, envolvendo muito mais comunicação do que computação.

Segundo Coulouris, Dollimore e Kindberg (2007, p. 17), a motivação para construir e usar sistemas distribuídos é proveniente do desejo de compartilhar recursos como memória,

espaço de armazenamento, tempo de processador e hardware periférico, aumentando a capacidade computacional e a disponibilização das informações.

Tanenbaum e Steen (2008, p. 2) afirma que o cerne de desenvolvimento de sistemas distribuídos é estabelecer como os diversos componentes (computadores) que compõem o sistema irão colaborar entre si. Observa-se que nenhuma premissa é adotada em relação ao tipo dos computadores. Dentro de um único sistema, eles podem variar desde *mainframes* até pequenos nós em redes de sensores. Da mesma forma, não há nenhuma premissa quanto ao modo como os computadores são interconectados.

As diferenças entre os vários computadores e o modo como eles se comunicam devem, em grande parte, ser ocultas aos usuários. Também deve ser fácil expandir um sistema distribuído. Essa característica é uma conseqüência direta de ter computadores independentes compondo o sistema. Um sistema distribuído deve estar permanentemente disponível, embora algumas partes possam estar temporariamente avariadas. Os usuários não devem perceber que uma parte está avariada, a não ser no caso de requisitar um recurso fornecido especificamente por aquela parte.

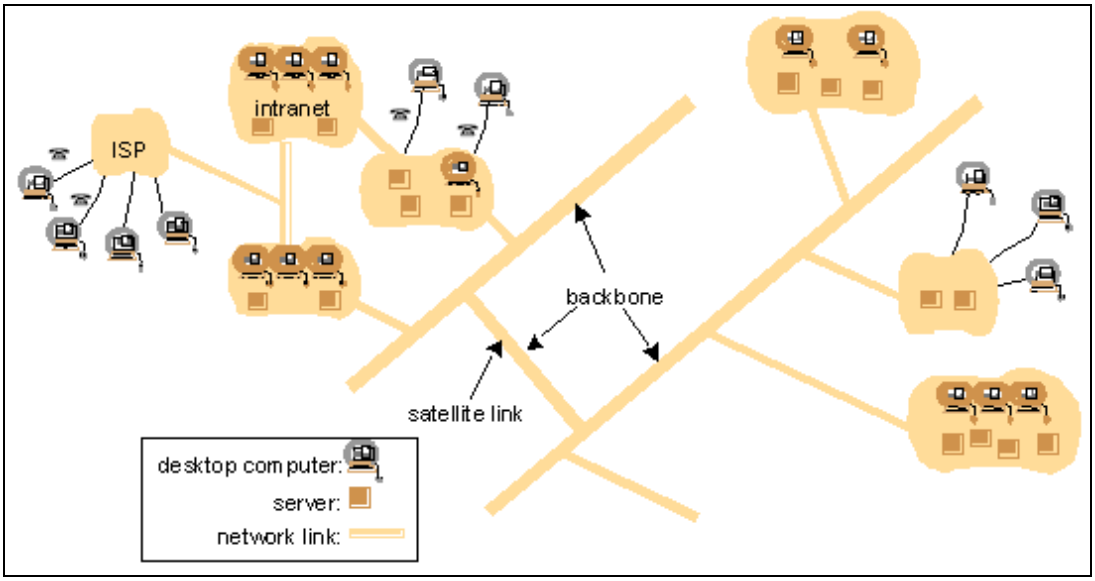
2.1.1 Exemplos de sistemas distribuídos

Os próximos tópicos apresentam alguns exemplos de sistemas distribuídos e suas principais características.

2.1.1.1 Internet

O maior exemplo de sistemas distribuídos é a Internet. Nela estão conectados milhões de computadores que trocam mensagens entre si para atender às requisições de suas aplicações. Os computadores que compõem a Internet são de muitos diferentes tipos, conectados à diferentes tecnologias de rede e com diversos sistemas operacionais. Esta heterogeneidade da infraestrutura é um dos maiores desafios, pois quando um novo computador é conectado ao sistema, deve ser capaz de trocar mensagens com todos os demais sem se preocupar com detalhes da construção de cada um. A Figura 1 mostra uma

configuração típica da Internet.



Fonte: Coulouris, Dollimore e Kindberg (2007, p. 17).

Figura 1 - Uma parte típica da Internet

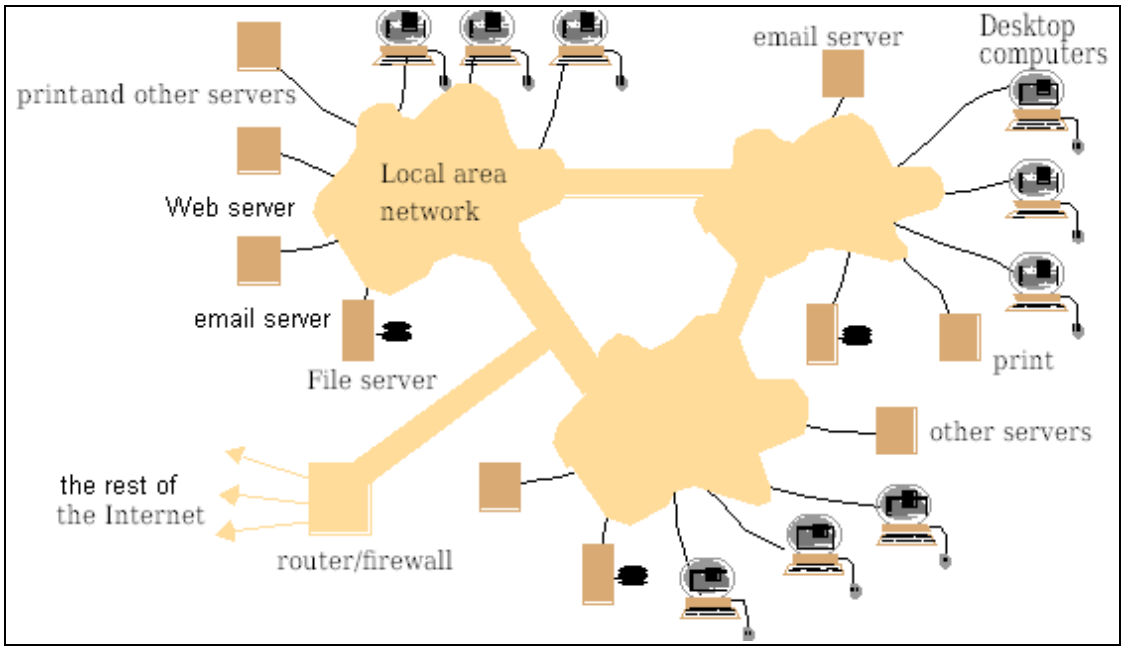
Os programas que estão em execução nos computadores conectados à internet interagem enviando mensagens através de um meio de comunicação comum. A Figura 1 mostra um conjunto de *intranets* - que são sub-redes interligadas por *backbones* (links de alta capacidade de transmissão). Os provedores de serviços de Internet, do inglês *Internet Service Providers* (ISP) são empresas que fornecem o acesso à Internet para usuários individuais ou organizações.

2.1.1.2 Intranet

Uma intranet é uma parte da Internet administrada separadamente, cujo limite pode ser configurado para impor planos de seguranças locais (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 18).

Uma intranet pode ser conectada à Internet por meio de um roteador, o qual permite aos usuários de dentro dessa intranet utilizarem serviços que são providos em outro lugar.

A Figura 2 apresenta a configuração de uma Intranet típica.



Fonte: Coulouris, Dollimore e Kindberg (2007, p. 18).

Figura 2 - Configuração de uma Intranet

Uma intranet é composta por várias LANs interligadas por conexões de *backbone*.

2.1.1.3 Computação móvel e ubíqua

Segundo Coulouris, Dollimore e Kindberg (2007, p. 19), os avanços tecnológicos de dispositivos e interligação em rede sem fio têm levado cada vez mais à interligação de equipamentos de computação pequenos e portáteis com sistemas distribuídos.

Os “computadores” utilizados pelos usuários da computação móvel são denominados dispositivos móveis. Um dispositivo móvel deve ser portátil, funcional e permitir fácil conectividade e comunicação com a rede e outros dispositivos (LEE; SCHNEIDER; SCHELL, 2005, p. 1). São exemplos destes dispositivos os *pagers*, telefones celulares, *Personal Digital Assitents* (PDA), *tablet* PCs e *laptops (notebooks)*.

A portabilidade de muitos desses dispositivos, junto com sua capacidade de se conectar convenientemente com redes em diferentes lugares, tornam a computação móvel possível. A computação móvel é a execução de tarefas de computação, enquanto o usuário está se deslocando de um lugar para outro ou visitando lugares diferentes de seu ambiente usual.

É notável a velocidade com que a computação móvel está se instalando nas

organizações. Scott (2007) afirma que “aparelhos como BlackBerries[7] e Treos[8] estão se tornando tão importantes no dia-a-dia corporativo, que as companhias precisam estender aplicações como *Enterprise Resource Planning (*ERP), sistemas de compras ou despesas a esses aparelhos”.

A computação ubíqua, ou pervasiva, é a utilização de vários dispositivos computacionais pequenos e baratos, que estão presentes nos ambientes físicos dos usuários, como suas casas, escritórios ou até vestuários (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 567). O termo “pervasivo” sugere que pequenos equipamentos de computação se tornarão tão entranhados nos objetos diários que mal serão notados. Um motorista, por exemplo, que utiliza um aplicativo rodando no celular para notificar o sistema da transportadora que uma determinada entrega foi realizada, pode fazê-lo sem ter consciência que seu celular é um “computador”, tão pouco da rede de computadores que precisou ser acionada para que sua notificação fosse processada.

Estes desafios são superados através da utilização de padrões, implementados através de *middlewares.*

Um *middleware* é uma camada adicional de software que fornece uma abstração de programação e o mascaramento da heterogeneidade das redes, do hardware, sistemas operacionais e linguagens de programação (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 29). Dependendo do paradigma da linguagem de programação utilizada, um *middleware* pode fornecer chamada de procedimento remoto ou chamada de método remoto.

Na comunicação com chamada de procedimento remoto, conhecida como RPC, o cliente envia uma requisição para o servidor, que a trata e devolve o resultado no estilo das linguagens de programação estruturada. Exemplos de implementação desta categoria são os Banco de Dados Cliente-Servidor, *Web services* e *Simple Object Access Protocol* (SOAP).

Na comunicação com chamada de método remoto, do inglês *Remote Method Invocation* (RMI), conhecidos como objetos distribuídos, o sistema cliente armazena uma referência remota para um objeto instanciado no servidor e faz chamadas aos métodos da interface pública deste objeto como se o mesmo estivesse na máquina local. CORBA, Java RMI e *Remoting* .NET são representantes dos *middlewares* para objetos distribuídos.

---

[7]Modelo de PDA da Research In Motion .

[8]Modelo de PDA da Handspring.

2.2 OBJETOS DISTRIBUÍDOS

Objetos distribuídos fornecem uma extensão ao modelo de objeto para torná-lo aplicável aos sistemas distribuídos (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 169).

2.2.1 Interface de objetos

A característica fundamental de um objeto é que ele encapsula dados, denominados estados e as operações executadas nesses dados, denominadas métodos. Os métodos são disponibilizados por meio de uma interface. É importante frisar que não há nenhum método "legal" pelo qual um processo possa acessar ou manipular o estado de um objeto, exceto pela invocação dos métodos da sua interface. Um objeto pode implementar várias interfaces e uma interface pode ser implementada por várias classes de objetos (AMBLER, 1998, p. 5).

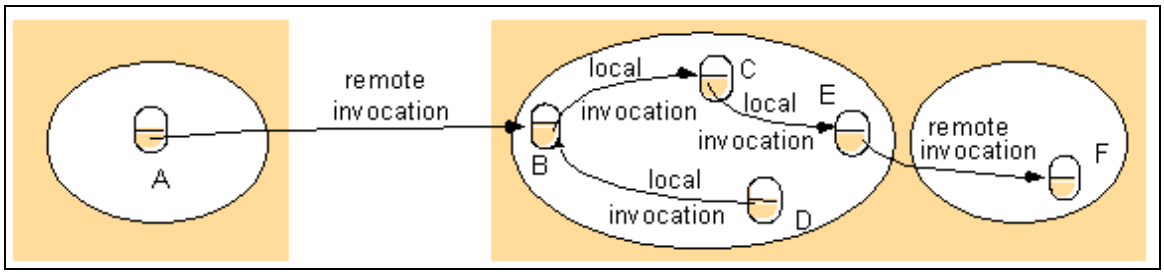
A separação entre interfaces e as classes de objetos que implementam essas interfaces é crucial para sistemas distribuídos. Ela nos permite colocar uma interface em uma máquina, enquanto o objeto em si reside em outra máquina. Essa organização é denominada objeto distribuído. Objetos distribuídos são um paradigma importante porque é relativamente fácil ocultar aspectos de distribuição sob a interface de um objeto (TANENBAUM; STEEN, 2008, p. 276).

Os *middlewares* de objetos distribuídos modernos procuram manter o máximo possível da semântica de objetos não distribuídos. Em outras palavras busca-se um alto grau de transparência na distribuição. Contudo, por questões de eficiência, a distribuição ainda é aparente em alguns aspectos (TANENBAUM; STEEN, 2008, p. 278). Um aspecto elementar mas decisivo, que impede a manutenção total da semântica não distribuída, é que métodos que nunca falham nas chamadas locais, como a soma de inteiros, por exemplo, podem falhar nas chamadas remotas por motivos relacionados ao fato do objeto invocado estar em um processo ou computador diferente do invocador. Por exemplo, o processo que contém o objeto remoto pode ter falhado, ou a conexão pode ter caído. Portanto, a invocação a método remoto deve ser capaz de lançar exceções como *timeouts* ou erros de entrada e saída (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 171).

Os conceitos fundamentais específicos para o modelo de objeto distribuído são o objeto remoto, a interface remota e a referência de objeto remoto (PEREIRA, 2003). As próximas seções discutem estes três conceitos e outros aspectos do modelo de objeto distribuído.

2.2.2 Objeto remoto ou servente

Objetos remotos, também denominados serventes, são objetos que podem receber invocações remotas. Num sistema de objetos distribuídos cada processo contém um conjunto de objetos, alguns dos quais podem receber invocações locais e remotas, enquanto outros objetos podem receber somente invocações locais, como mostra a Figura 3. As invocações entre objetos em diferentes processos, seja no mesmo computador ou não, são conhecidas como invocações à métodos remotos (RMI). As invocações a métodos entre objetos no mesmo processo são chamadas de invocações à métodos locais (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 169).



Fonte: Coulouris, Dollimore e Kindberg (2007, p. 169).

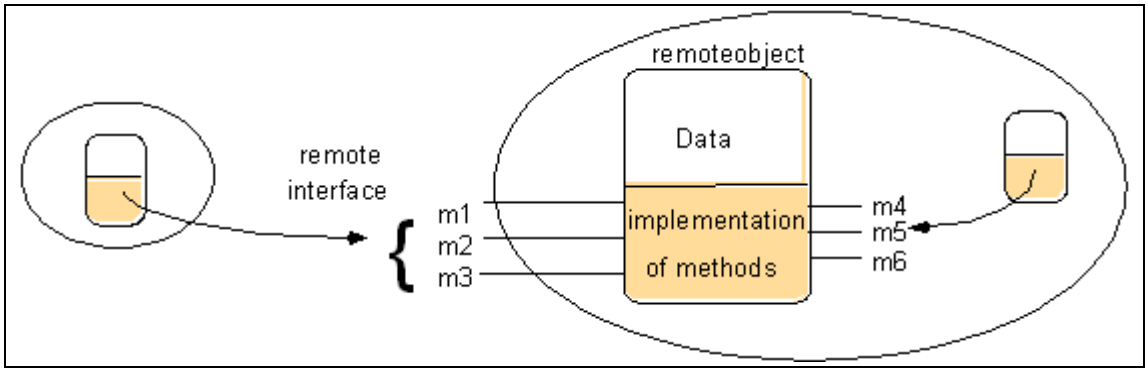
Figura 3 - Invocação a métodos locais e remotos

Na Figura 3, B e F são objetos remotos. Todos objetos locais podem receber invocações locais de outros objetos que contenham referências a eles. Por exemplo, o objeto C deve ter uma referência ao objeto E para poder invocar um de seus métodos públicos. Um objeto não têm como referenciar um objeto remoto diretamente. Para isto, dois componentes dos objetos distribuídos precisam ser acionados. A interface remota e a referência de objeto remoto.

2.2.3 Interface remota

No modelo de objetos distribuídos, uma interface remota especifica os métodos de um objeto que estão disponíveis para invocação por parte dos objetos de outros processos, definindo os tipos dos argumentos de entrada e saída de cada um deles (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 166). Objetos de outros processos só podem invocar os métodos pertencentes à interface remota, como mostrado na Figura 4.

Uma característica importante à respeito dos objetos remotos é que seu estado não é distribuído: ele reside em uma única máquina. Somente as interfaces remotas implementadas pelo objeto remoto são disponibilizadas em outras máquinas (TANENBAUM; STEEN, 2008, p. 269).



Fonte: Coulouris, Dollimore e Kindberg (2007, p. 170).

Figura 4 - Um objeto remoto e sua interface remota

No exemplo da Figura 4 `remoteObject` posssui seis métodos (m1, m2, m3, m4, m5 e m6). No entanto, apenas m1, m2 e m3 pertencem à interface remota e podem ser invocados por objetos de outros processos.

2.2.4 Referência de objeto remoto

Segundo Coulouris, Dollimore e Kindberg (2007, p. 170), “a noção de referência de objeto é estendida para permitir que qualquer objeto que possa receber RMI tenha uma referênca de objeto remoto”. Uma referência de objeto remoto é uma instância de uma classe que implementa a interface remota do objeto e que pode ser usada por todo um sistema distribuído para se referir a um objeto único em particular.

Quando um cliente se vincula a um objeto distribuído, uma instância da implementação da sua interface remota, denominada *proxy*, é carregada no espaço de endereçamento do cliente.

Um *proxy*, que é a referência para o objeto remoto, é responsável por tornar a invocação a método remoto transparente para os clientes, comportando-se como um objeto local para o invocador; mas em vez de executar uma invocação local, ele a encaminha em uma mensagem para o um objeto remoto. Ele oculta os detalhes da referência do objeto remoto, do empacotamento de argumentos, do desempacotamento dos resultados e do envio e recepção das mensagens. Existe uma implementação do *proxy* para cada classe de objeto remoto disponibilizada por um processo. Um *proxy* implementa todos os métodos da interface remoto do objeto remoto. Porém, o *proxy* implementa os métodos da interface de uma forma diferente. Cada método do *proxy* empacota uma referência para o objeto alvo, o *methodID* e seus argumentos em uma mensagem de requisição e a envia para o objeto alvo. A seguir, espera pela mensagem de retorno, quando a recebe, desempacota e retorno os resultados para o invocador (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 175).

2.2.5 Servidores de objetos

Um servidor de objetos é um servidor configurado para suportar objetos distribuídos. A diferença entre um servidor de objetos e outros servidores, é que um servidor de objetos, por si só, não fornece serviço algum. Os serviços são implementados pelos objetos que residem no servidor. Em essência, o servidor fornece apenas os meios de invocar objetos locais, com base em requisições de clientes remotos. Por isso, é relativamente fácil mudar serviços apenas com adição ou remoção de objetos (TANENBAUM; STEEN, 2008, p. 272).

2.2.6 Despachantes e Esqueletos

Um servidor de objetos tem um despachante (*dispatch)* e um esqueleto (*skeleton)* para cada classe que representa um objeto remoto. O despachante recebe a mensagem da requisição e utiliza o *methodID* para selecionar o método apropriado no esqueleto, repassando

a mensagem de requisição. O despachante e o *proxy* usam o mesmo *methodID* para os métodos da interface remota. O esqueleto implementa os métodos da interface remota mas de uma forma diferente dos métodos implementados no servente que personifica o objeto remoto. Um método de esqueleto desempacota os argumentos na mensagem de requisição e invoca o método correspondente no servente. O esqueleto espera que a requisição termine e, em seguida, empacota o resultado, junto com as exceções, em uma mensagem de resposta que é enviada para o método do *proxy* que fez a requisição (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 175).

2.2.7 Agente de requisição de objeto

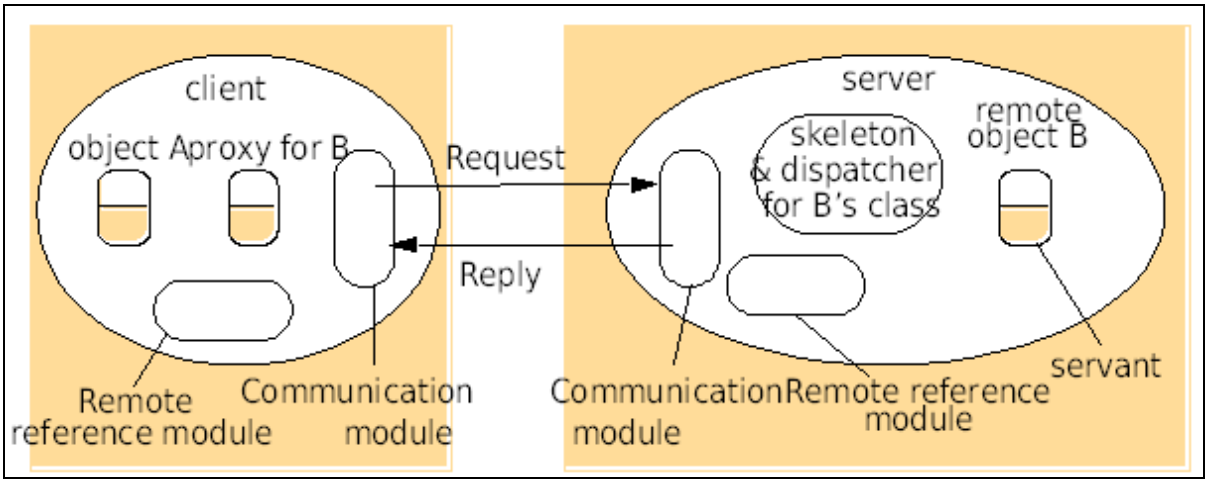
Agente de requisição de objeto, do inglês *Object Request Broker* (ORB) é uma metáfora introduzida pelo *Object Management Group* (OMG), para denominar a classe de objeto responsável por ajudar um processo cliente a invocar um método em um objeto remoto. Essa função envolve localizar o objeto, invocá-lo e então comunicar a requisição do cliente ao objeto remoto que a executa e responde (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 711). Em outras palavras, o ORB cuida dos detalhes de baixo nível envolvidos no caminho de uma requisição do cliente até o objeto remoto e do retorno da resposta para o seu destino. Há duas implementações de ORB, uma no servidor e outra no cliente que realizam operações complementares entre si (ORB BASICS, 2007).

2.2.8 Módulo de referência remota

O módulo de referência remota é responsável pela transformação entre referências de objeto local e remoto e pela criação de referência de objeto remoto. Há uma instância do módulo de referência remota em cada processo e a mesma contém uma tabela de objetos remotos para registrar a correspondência entre as referências de objeto local desse processo e as referências de objeto remoto (que abrangem todo o sistema). A tabela inclui:

a) uma entrada para todos os objetos remotos mantidos pelo processo. Por exemplo, na Figura 5 , o objeto remoto B estará registrado na tabela do servidor;

b) uma entrada para cada *proxy* local. Por exemplo, na Figura 5, o *proxy* B estará registrado na tabela do cliente.



Fonte: Coulouris, Dollimore e Kindberg (2007, p. 174).

Figura 5 - *Proxy* e esqueleto na invocação a método remoto

Quando um objeto remoto precisa ser passado como argumento ou resultado pela primeira vez, o módulo de referência remota cria uma referência de objeto remoto, a qual adiciona em sua tabela. Quando uma referência de objeto remoto chega em uma mensagem de requisição ou resposta, é solicitada ao módulo de referência remota a correspondente referência ao objeto local, a qual pode se referir a um *proxy* ou objeto remoto (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 175).

2.2.9 Módulo de Comunicação

Dois módulos de comunicação cooperam para executar o protocolo requisição-resposta, o qual transmite mensagens entre o cliente e o servidor. O módulo de comunicação utiliza os três primeiros elementos, que especificam o tipo de mensagem, o identificador da requisição e a referência remota do objeto invocado.

O módulo de comunicação do servidor seleciona o despachante para a classe do objeto remoto a ser invocado, passando sua referência local, obtida a partir do módulo de referência remota através do identificador do objeto remoto dado pela mensagem de requisição (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 174).

2.2.10 Representação Externa de Dados

As informações armazenadas nos programas em execução são representadas como estruturas de dados, enquanto que as mensagens são seqüências puras de bytes. Independente da forma de comunicação usada, as estruturas de dados devem ser convertidas em seqüências de bytes antes da transmissão e reconstruídos na sua chegada (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 135).

Nem todos computadores armazenam as informações no mesmo formato. A representação de números inteiros, números em ponto flutuante, datas e seqüências de caracteres diferem entre as arquiteturas de computador. Para que informações possam ser trocadas entre computadores que com diferentes formatos para os dados internos, é necessário que um padrão comum seja determinado, para formatação das mensagens transmitidas. Um padrão aceito para representação de estruturas de dados e valores é chamado de representação externa de dados, do inglês *eXternal Data Representation* (XDR) (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 137).

O processo de transformar um conjunto de itens de dados internos em XDR para transmissão é chamado de empacotamento, do inglês *marshalling.* Desempacotamento (*unmarchalling)* é o processo inverso de transformar um XDR no conjunto de itens de dados internos compatível com o computador que recebe a mensagem.

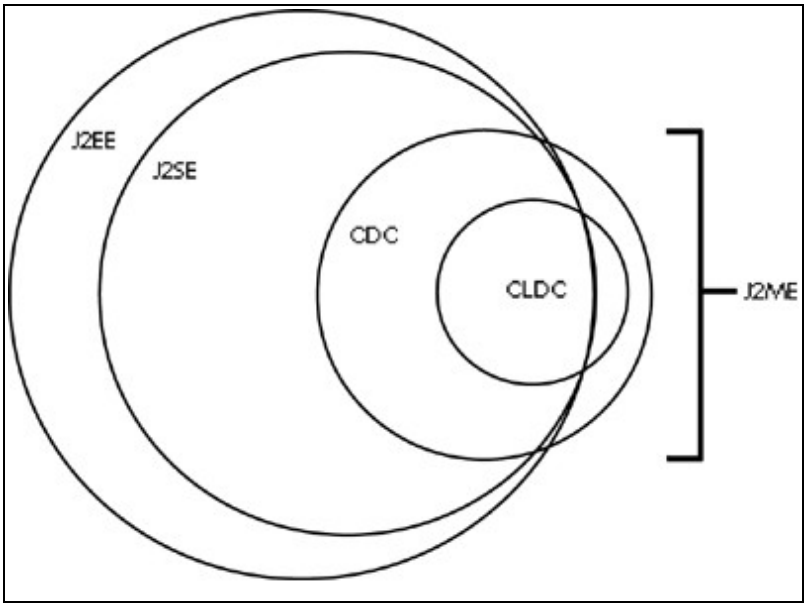
Um XDR bastante popular é o XML, que define um formato textual para representar dados estruturados. Originalmente, XML se destinava a documentos contendo dados estruturados textuais auto-descritivos; por exemplo documentos web. Mas agora também é usuado para representar dados enviados em mensagens trocadas por clientes e servidores em serviços web (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 137).

O processo de XDR é uma atividade de responsabilidade da camada *middleware*, sem nenhum envolvimento por parte do programador de aplicação. Isso porque o processo de empacotamento exige a consideração de todos os mínimos detalhes da representação dos componentes primitivos de objetos compostos e é bastante propenso a erros se executado manualmente (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 137).

2.3 JAVA MICRO EDITION

A tecnologia Java é composta por uma linguagem de programação e diversas plataformas especializadas. A plataforma Java é uma plataforma de software que roda sobre diferentes plataformas de hardware. Como as plataformas de hardware variam muito em sua capacidade de armazenamento, memória, tecnologia de rede e capacidade de processamento, diferentes plataformas Java são disponíveis para atender o desenvolvimento de aplicações para os diferentes ambientes. Cada plataforma é baseada numa máquina virtual Java, do inglês Java Virtual Machine (JVM), que é portável para um ambiente de hardware específico. Isso significa, por exemplo, que um software desenvolvido para computadores do tipo *Desktop*, rodará em qualquer computador desta linha, independente da tecnologia de hardware (PC, Mac, ou outro) ou do sistema operacional (Windows, Linux, Solaris, ou outro) (JAVA TECHNOLOGY OVERVIEW, 2008).

Java divide suas plataformas em três edições: a primeira e mais abrangente é a JEE, focada em aplicações corporativas de grande porte; a segunda é a JSE, focada nos computadores *desktop;* e a menos abrangente é a JME, para dispositivos móveis PDA's, celulares, e eletrônicos em geral. Aplicações desenvolvidas para uma configuração menos abrangente, rodará numa JVM mais abrangente, mas o contrário não funcionará. A Figura 6 apresenta a organização da tecnologia Java em suas diferentes edições.



Fonte: Barr (2002).

Figura 6 - A edições e configurações do Java

A edição JME se destaca por ser extremamente limitada em comparação às demais.

Tanto que sua JVM têm uma designação própria a *Kilobyte Virtual Machine* (KVM). Um número significativo de funcionalidades do JSE foi eliminado na especificação de JME, seja devido às limitações de hardware, seja porque sua presença implicaria em problemas de segurança (MAHMOUD 2001). JME apresenta duas configurações: *Connected Device Configuration* (CDC) e *Connected Limited Device Configuration* (CLDC). A maioria dos dispositivos móveis suporta CLDC. Dentre as limitações apresentadas por CLDC, duas são especialmente críticas para a implementação de objetos distribuídos nesta plataforma:

a) sem finalização: CLDC não inclui o método `Object.finalize`, assim não se pode executar operações sobre o objeto antes dele ser eliminado pelo coletor de lixo, do inglês Garbage Collector (GC);
b) sem suporte para reflexão: por não suportar reflexão, CLDC também não oferece suporte ao Java RMI ou mesmo serialização de objetos.

2.4 *MIDDLEWARES*

A seguir são discutidos exemplos de *middlewares* para implementação de sistemas distribuídos. Primeiramente são apresentados os web services, que é um exemplo de *middleware* para requisição de procedimentos distribuídos. Em seguida é apresentado o Java RMI, representante de *middleware* para objetos distribuídos.

2.4.1 Web services

Web services surgiram com a idéia de resolver os problemas de incompatibilidade entre os protocolos, uso de portas e formatos binários rejeitados pelos *firewalls*,dentre outros problemas apresentados em software projetado com arquitetura em camada (ROCHA, 2007, p.16). Trata-se de um *middleware* que provê uma série de interfaces para permitir a comunicação entre aplicações na Web (WEB SERVICES ACTIVITY, 2002). Web services funciona sobre SOAP que é um protocolo simples para troca de mensagens entre sistemas distribuídos, utilizando XML, dentro da filosofia RPC (EHNEBUSKE et al., 2000).

Conforme Rocha (2007, p.17), os principais benefícios dos Web services são:

a) protocolos baseados no padrão XML, permitindo a geração automática do código, tanto no cliente, quanto no servidor;
b) utilização de protocolos baseados em texto, o que permite tráfego suave através de *firewalls* que fazem verificações de pacotes;
c) utilização da porta 80 do protocolo HTTP, o que permite transportar as chamadas ao serviço sem que o firewall bloqueie esta porta.

Com SOAP, o cliente realiza a requisição para o servidor e recebe suas respostas através de estruturas XML. Esta abordagem provê grande flexibilidade ao protocolo, pois o XML descreve perfeitamente os dados em tempo de execução e evita problemas causados por eventuais mudanças na assinatura dos procedimentos.

Normalmente SOAP é utilizado sobre o HTTP. Esta característica facilita a implantação, pois o SOAP pode ser implementado no próprio servidor web, utilizando os mesmos serviços de autenticação e criptografia. A desvantagem do SOAP é a quantidade de informação trafegada em cada requisição para descrever o XML. A título de exemplo, o Quadro 1 apresenta a estrutura XML enviada para o servidor para fazer uma chamada do tipo `double x = getLastPrice ("produto1").`

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastPrice xmlns:m="Some-URI">
           <symbol>produto1</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Quadro 1 - XML para a chamada de procedimento remoto

A estrutura da requisição do Quadro 1 apresenta a estrutura das informações transmitidas entre os processos, para realizarem requisições utilizando SOAP.

2.4.2 Java RMI

O Java RMI permite o desenvolvimento de aplicações Java distribuídas, de forma que os métodos dos objetos Java remotos possam ser invocados a partir de aplicações rodando em outras JVM (JAVA REMOTE METHOD INVOCATION SPECIFICATION, 2004).

Em Java RMI o código cliente invoca os métodos de um objeto denominado *stub.* Um *stub* implementa a mesma interface do objeto remoto, sendo a sua função abstrair os detalhes do envio das requisições para o objeto remoto e do recebimento dos resultados. Como o *stub* é gerado a partir do objeto remoto, os detalhes do formato das mensagens de invocação são determinados em tempo de compilação. Esta abordagem permite que na invocação, sejam enviados apenas os parâmetros do método chamado, eliminando a necessidade de pesadas estruturas de metadados, como ocorre no SOAP.

É importante observar que nem todas as edições de Java suportam RMI. O Java é organizado em três edições: a JSE, que é a edição mais popular, utilizada no desenvolvimento de aplicações *desktop;* JEE, utilizada nos sistemas dos servidores de aplicações corporativas e o JME, aplicado ao desenvolvimento de software para dispositivos móveis. O JME (que nas versões anteriores ao Java 5 era chamado de J2ME), por sua vez, possui duas configurações: CDC, aplicável apenas em dispositivos móveis com elevada capacidade computacional como alguns modelos de *Pocket PC's* e *Tablet* PC's; CLDC, utilizada no desenvolvimento de aplicações para celulares e a maioria dos demais dispositivos móveis. Não existe Java RMI para CLDC. Além disso, Java RMI não fornece suporte para programas escritos em outras linguagens. Outra característica de Java RMI é que esta tecnologia utiliza um *range* dinâmico de portas para estabelecer as conexões (JAVA REMOTE METHOD INVOCATION, 2008), o que as faz serem normalmente bloqueadas pelos *firewalls*.

2.5 COMPUTAÇÃO MÓVEL

Computação móvel é a execução de tarefas computacionais, enquanto o usuário está em deslocamento ou fora de seu local habitual (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 19).

Segundo Loureiro *et al*. (2007), dispositivos computacionais móveis não são simples organizadores pessoais. Com o desenvolvimento tecnológico de fabricação de circuitos integrados que ocorreu, principalmente, durante a última década, tem sido possível a fabricação de dispositivos computacionais que possuem um novo paradigma: o paradigma da mobilidade. Esse paradigma está mudando a forma como faz-se diversas atividades diárias ou não deseja-se ficar “preso” a uma infra-estrutura fixa de comunicação de dados.

O paradigma de computação móvel é uma evolução natural quando analisa-se os

outros paradigmas que foram e são usados até hoje.

Na década de 1960, o paradigma que prevaleceu foi o de processamento em lote (*batch*), onde o usuário preparava, submetia e recebia seu *job* sem ter nenhum contato com o ambiente computacional. Na década de 1970 surge o sistema computacional multitarefa e o teleprocessamento. O usuário passa a ter acesso ao computador através de terminais remotos. É a época do surgimento do Centro de Processamento de Dados (CPD), que ainda é utilizado até hoje. No início da década de 1980 começa a ser difundido em larga escala o computador pessoal, que passa a ser o paradigma dominante daí em diante, principalmente com o desenvolvimento do hardware associado a esse tipo de computador. Na década de 1990, os computadores pessoais passam a ser utilizados em larga escala em todas as atividades humanas, com a característica de estarem conectados a alguma rede, principalmente a Internet, que passa a ser a infra-estrutura de rede de abrangência global mais utilizada pelas pessoas.

Computação móvel é um novo paradigma computacional que tem como objetivo prover ao usuário acesso permanente a uma rede fixa ou móvel independente de sua posição física. É a capacidade de acessar informações, aplicações e serviços a qualquer lugar e a qualquer momento. Este paradigma também recebe o nome de computação ubíqua ou computação nômade. Existem três elementos que caracterizam e compõem a computação móvel: o tipo e capacidade de processamento do dispositivo portátil, a mobilidade do usuário e da unidade móvel, e a comunicação com outro elemento computacional através de um canal de comunicação sem fio.

2.6 HTTP

O protocolo HTTP é uma arquitetura de sistema cliente-servidor, com regras padrão para interação, por meio das quais os clientes buscam recursos dos servidores web (COULOURIS; DOLLIMORE; KINDBERG, 2007, p. 150).

Segundo Junior (2001, p. 6), o HTTP é o responsável pela maioria das transações realizadas na Internet e também é o grande responsável pelo sucesso da web.

Connolly (1999), define HTTP como um protocolo de aplicação para sistemas de informação hipermídia distribuídos e colaborativos. HTTP é utilizado pela *web* deste 1990 e atualmente está na versão 1.1.

O HTTP é um protocolo do tipo requisição/resposta. Um cliente envia uma requisição para o servidor o seu identificador uniforme de recurso, do inglês *Uniform Resource Identifier* (URI) e a versão do protocolo, seguido por uma mensagem, formatada de acordo com as extensões multi função para mensagens de internet, do inglês *Multipurpose Internet Mail Extensions* (MIME). O servidor responde com uma linha de status que inclui a versão do protocolo, um código de sucesso ou erro uma mensagem no formato MIME.

É comum que o protocolo HTTP funcione sobre conexões *Transmission Control Protocol/Internet Protocol* (TCP/IP). A sua porta padrão é TCP 80, mas outras portas podem ser usadas. Não há restrições para que HTTP seja implementado sobre qualquer outro protocolo na Internet. Qualquer protocolo que disponibilize um transporte confiável pode ser utilizado.

Nas versões anteriores à 1.1, HTTP criava uma nova conexão para cada requisição do cliente para o servidor. Isto criava uma sobrecarga nos servidores e congestionava a Internet. A partir da versão 1.1 o protocolo passou a trabalhar com conexões persistentes por *default*. Assim, salvo configuração em contrário, o cliente deve assumir que o servidor irá manter uma conexão persistente.

A chave do sucesso do HTTP é a sua simplicidade. Ele especifica as mensagens envolvidas em uma troca de requisições e respostas através de um conjunto fixo de métodos que são aplicáveis a todos os seus recursos. Detalhes sobre o recurso e tipo de conteúdo são incluídos na mensagem MIME através de meta-dados. Esta abordagem permite que o cliente realize uma requisição sem saber previamente se a resposta será um texto, uma imagem ou um arquivo executável, por exemplo. A própria mensagem da resposta lhe revelará estas informações.

2.7 SERVLETS

Segundo Jendrock *et al* (2007), desde que a web passou a ser utilizada para disponibilizar serviços, os provedores identificaram a necessidade de oferecer conteúdo dinâmico. Inicialmente as portas de comunicação comum do inglês Common Gateway Interface (CGI) eram a principal tecnologia utilizada para gerar conteúdo dinâmico. Embora amplamente utilizado, CGI tem uma série de inconvenientes como dependência de plataforma e escalabilidade difícil. A tecnologia de Servlets foi desenvolvida para superar as limitações

apresentadas por CGI.

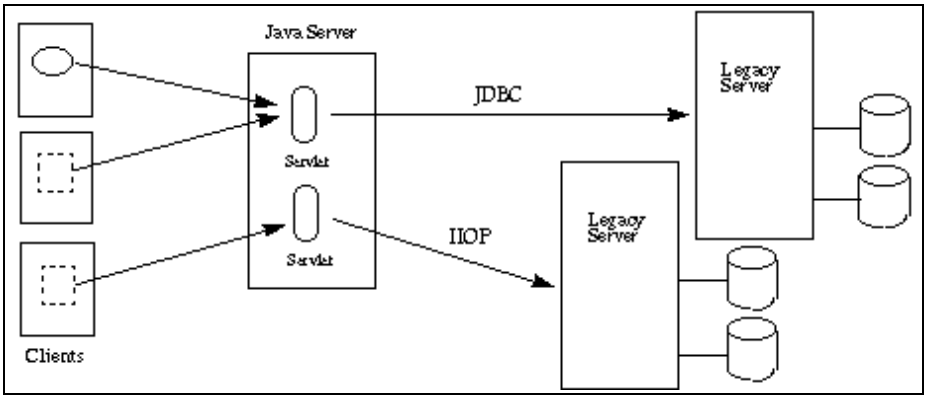
The Java Servlet API White Paper (1998) diz que *servlets* são componentes do lado do servidor independentes de protocolo e plataforma de hardware, escritos em Java, que estendem de forma dinâmica os servidores de aplicações Java. Eles provêm um *framework* de uso geral para construção de serviços, utilizando o paradigma requisição-resposta. Sua aplicação inicial foi prover o acesso seguro via web aos dados que são apresentados através de páginas HTML.

Servlets podem processar requisições de protocolos seguros como *Secure Sockets Layer* (SSL), de forma a garantir a identidade dos processos clientes e a inviolabilidade das informações transmitidas nas requisições e respostas.

Como são totalmente baseados em Java, aplicações que utilizam s*ervlets* no servidor possuem a garantia de que violação de memória e violação de tipos de dados não são possíveis. Esta garantia diminui significativamente os riscos de *crash* nos servidores.

Em termos de performance, *servlets* possuem a característica de não requerer a criação de um novo processo para cada requisição. Vários s*ervlets* rodam em paralelo dentro do mesmo processo no servidor.

Uma aplicação comum para *servlets* é a geração de páginas HTML dinâmicas. Neste contexto, *servlets* são a alternativa Java para tecnologias como CGI ou *Hypertext Preprocessor* (PHP). Em outro contexto, eles podem atender as requisições de qualquer aplicação projetada para conversar com o servidor no modelo requisição-resposta sobre HTTP. Neste caso é comum desenvolver em *servlets* a segunda camada em aplicações de três camadas, como mostra a Figura 7.



Fonte: The Java Servlet API White Paper (1998).

Figura 7 - Servlets em aplicações de 3 camadas

Nesto exemplo da Figura 7, *servlets* fazem a integração entre a primeira camada (clientes) e a terceira (sistemas legados).

2.8 REFLEXÃO COMPUTACIONAL

Segundo Pavan (1998), a reflexão computacional permite fazer computações sobre uma computação, com o objetivo de alterar e adaptar sistemas de forma dinâmica. Define uma arquitetura em níveis, denominada arquitetura reflexiva, composta por um meta-nível, onde se encontram as estruturas de dados e as ações a serem realizadas sobre o sistema objeto, localizado no nível base.

Em linguagens orientadas a objetos, a reflexão computacional é realizada através de meta-classes ou meta-objetos, que se distinguem por sua abrangência; a reflexão realizada por meio de uma meta-classe altera todas as instâncias da classe, enquanto que a reflexão realizada por meio de um meta-objeto refere-se a uma única instância. Em ambas as formas, o objetivo é fornecer informações sobre a representação dos métodos e das propriedades das classes ou objetos da aplicação .

Reflexão computacional pode ser implementado em diferentes modelos, destacando-se o modelo de meta-classes e de meta-objetos.

O modelo de meta-classes é adotado em linguagens que estruturam as suas classes a partir de meta-classes. Este modelo é conhecido como reflexão estrutural, visto que permite obter informações e realizar transformações sobre a estrutura estática de uma classe.

No modelo de meta-objetos, a cada objeto da aplicação pode ser associado um meta-objeto que representa aspectos estruturais e comportamentais de um objeto, a ele conectado. As classes do objeto e do meta-objeto são distintas, tornando esse modelo mais flexível do que o modelo de meta-classes, por ser a associação realizada através de objetos e não de classes.

Para Muhammad (2003), outra peculiaridade de linguagens reflexivas é o fato de que torna-se possível manipular as descrições de objetos de forma similar à que os objetos em si são manipulados,isto é, aquilo que tradicionalmente é visto como classe torna-se também objeto. Este processo no qual entidades abstratas são convertidas a entidades concretas, passíveis de manipulação é chamado de reificação. O objeto decorrente deste processo possui uma classe (isto é, uma meta-classe) passível de ser reificada. Torna-se claro que, neste contexto, todas as entidades do programa podem ser encaradas como objetos.

2.9 TRABALHOS CORRELATOS

A seguir são apresentados dois trabalhos desenvolvidos com o objetivo de permitir a implementação de sistemas distribuídos em dispositivos móveis.

2.9.1 *Middleware* para Ambientes Pervasivos (MAP)

MAP permite o desenvolvimento de aplicações distribuídas com arquitetura orientada à serviços em dispositivos móveis. Segundo Loureiro (2006), "esse *middleware*, fornece suporte ao desenvolvimento de aplicações e serviços, à publicação, descoberta e utilização desses últimos em ambientes pervasivos e ainda à aquisição de informações de contexto".

Voltado para dispositivos que possuam JVM nas configurações CLDC ou CDC, MAP possui um mecanismo que permite sua atualização em tempo de execução, utilizando-se da facilidade de código móvel do Java.

Loureiro (2006) afirma que "MAP considera qualquer dispositivo como cliente ou provedor de recursos em potencial. Como conseqüência, cria-se um repositório dinâmico de recursos, todos disponíveis aos usuários móveis através de seus dispositivos".

MAP utiliza-se do protocolo SOAP para troca de mensagens, provendo acesso à procedimentos distribuído, não suportando a construção de sistemas baseados em objetos distribuídos.

2.9.2 Chamada ReMota para *j2mE* (RME)

RME fornece aos desenvolvedores JME com configuração CLDC, uma alternativa para a implementação da invocação remota de métodos, originalmente não atendida por esta configuração do Java. Segundo Pereira (2003), RME é "[...] uma plataforma de *middleware* que fornece aos desenvolvedores de aplicações móveis um serviço de invocação remota de métodos. RME foi desenvolvido sobre o perfil MIDP/CLDC da plataforma JME [...]". Pereira (2003) diz também que "[...]não existe ainda nenhum sistema desse tipo para CLDC cuja

implementação esteja consolidada".

Além de ser a única alternativa de RMI para dispositivos móveis identificada, a biblioteca cliente de RME ocupa apenas 34,5 Kbytes, o que a viabiliza para quase todos os modelos de dispositivos móveis existentes.

As limitações de RME são que fornece suporte apenas para programas escritos em Java e opera sobre um *range* de portas para estabelecer as conexões, o que dificulta sua implantação devido aos bloqueios dos *firewalls* das empresas.

3 DESENVOLVIMENTO

Este capítulo detalha as etapas do desenvolvimento do *middleware* HMI. São ilustrados os principais requisitos, a especificação, a implementação (mencionando técnicas e ferramentas utilizadas, bem como a operacionalidade do *middleware*) e por fim são listados resultados e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O objetivo deste trabalho é desenvolver um *middleware* que permita a construção de sistemas distribuídos orientados a objetos, utilizando HTTP como protocolo de comunicação. Para isto, foram identificados os seguintes requisitos:

a) permitir o registro de objetos distribuídos no servidor (Requisito Funcional – RF);
b) permitir que o cliente localize os objetos distribuídos registrados no servidor (RF);
c) retornar uma referência remota quando a chamada para um método remoto retornar um objeto remoto (RF);
d) permitir que o cliente libere do servidor os objetos que não serão mais utilizados (RF);
e) liberar do servidor os objetos que não tenham mais referência em nenhum cliente (RF);
f) ocupar menos de 40 Kbytes nos clientes (Requisito Não Funcional – RNF);
g) possuir uma especificação que permita a implementação do cliente em diferentes linguagens de programação (RNF);
h) gerar stubs e *skeletons* para os objetos remotos (RF).

3.2 ESPECIFICAÇÃO

A especificação do presente trabalho foi desenvolvida utilizando a notação UML (BOOCH; RUMBAUGH; JACOBSON, 2002) em conjunto com apoio da ferramenta

Enterprise Architect (EA) (ENTERPRISE ARCHITECT, 2008), onde serão apresentados diagramas de casos de uso, classes e seqüência. Alguns diagramas estão em sua forma resumida para melhor visualização, mas sem comprometer o entendimento do trabalho.

Para apresentar a especificação o *middleware* foi dividido em três módulos:

a) Módulo de *toolkits:* utilizado durante o desenvolvimento da aplicação que utiliza o *middleware*, gera o *stub* e s*keleton* que permitirão a invocação de método remoto, a partir de uma classe de objeto remoto implementada em Java;
b) Módulo Servidor: conjunto de classes que são disponibilizadas num *servlet container* e têm como finalidade possibilitar que objetos remotos tenham seus métodos invocados através de sua interface remota;
c) Módulo Cliente: classes que devem ser distribuídas com a aplicação que utiliza o *middleware* e têm a função de permitir que a mesma consiga tratar uma interface remota e realizar chamadas aos objetos remotos instanciados no módulo servidor.

A Figura 8 apresenta os principais casos de uso atendidos pelo *middleware.*

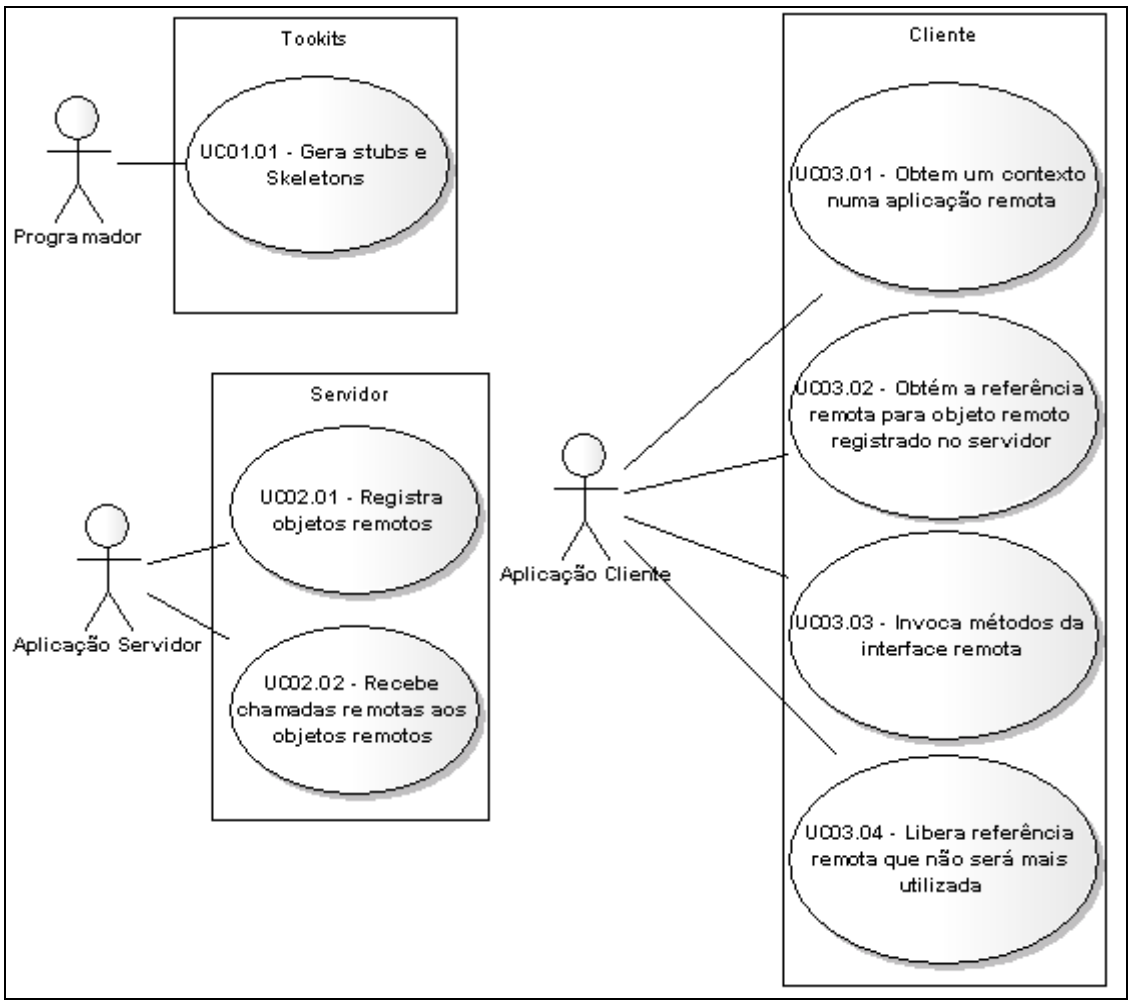


Figura 8 - Principais casos de uso do *middleware*

Para apresentar como as classes da *middleware* estão estruturadas e relacionadas

utilizou-se o diagrama de classes. A seguir serão explanados as classes dos módulos que compõem o *middleware* HMI.

3.2.1 Módulo servidor

O módulo servidor tem como objetivo armazenar os objetos remotos que terão os métodos da sua interface remota, invocados pela aplicação cliente. A Figura 9 mostra o diagrama de classes do módulo servidor.

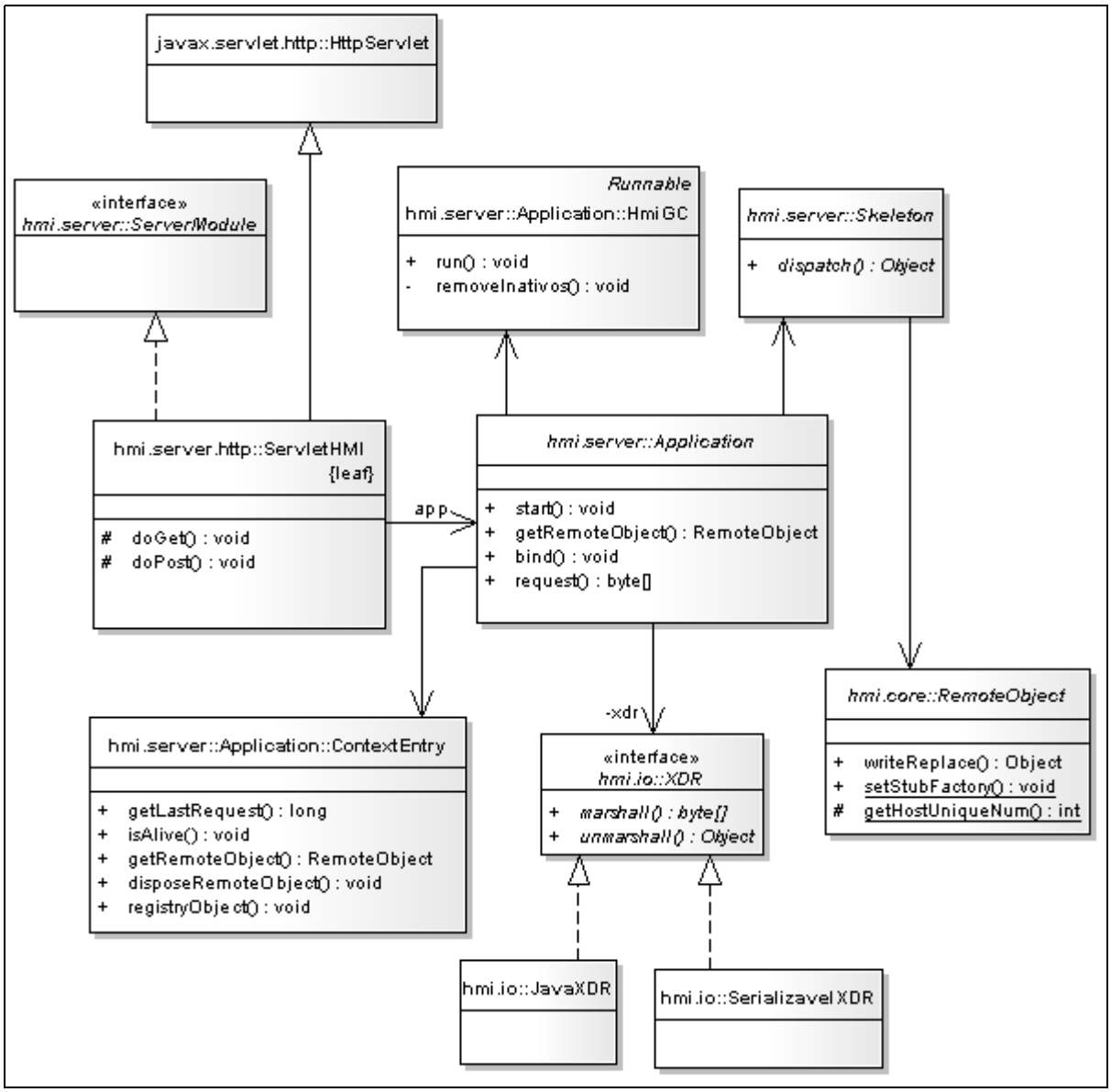


Figura 9 - Diagrama de classes do módulo servidor

O Quadro 2 apresenta as responsabilidades das classes e interfaces do módulo servidor.

Classe	**Responsabilidade**
`hmi.server.ServerModule`	Interface do *middleware* implementada pelo módulo de comunicação do servidor. Não possui métodos. Sua função é permitir que o *middleware* identifique a classe que implementa o módulo.
`hmi.server.http.ServeletHMI`	Implementação *default* do módulo de comunicação dos servidor sobre HTTP, fornecida pelo *middleware*. Do ponto de vista da tecnologia Java é um *servlet* (estende `javax.servlet.http.HttpServlet`).
`hmi.server.Application`	Classe abstrata que deve ser estendida pelo programador da aplicação e ter seu método abstrato `start` implementado. O método `start` é invocado pelo módulo de comunicação do servidor quando a aplicação é carregada.
`hmi.serverApplication::ContextEntry`	Classe interna de `hmi.server.Application` responsável por armazenar o contexto dos clientes remotos.
`hmi.serverApplication::HmiGC`	Classe interna de `hmi.server.Application`, localiza instâncias de `Hmi.serverApplication::ContextEntry` cujos clientes remotos não estão mais conectados ao servidor, para que os mesmos sejam desalocados.
`hmi.server.HmiSkeleton`	Classe abstrata que deve ser estendida pelas classes de *Skeletons* dos objetos remotos.
`hmi.core.RemoteObject`	Classe abstrata que deve ser estendida pelas classes de objetos remotos.
`hmi.io.XDR`	Especifica a interface pública das classes que realizam XDR.
`hmi.io.JavaXDR`	Implementação de XDR baseada na serialização nativa da tecnologia Java.
`Hmi.io.SerializavelXDR`	Implementação de XDR que serializa objetos de classes que implementam hmi.io.Serializavel.

Quadro 2 - Responsabilidades das classes do módulo servidor

3.2.2 Módulo Cliente

O módulo cliente tem as seguintes responsabilidades:

a) atender a solicitação de localização de objetos remotos e disponibilizar a sua referência remota à aplicação que utiliza o *middleware;*
b) solicitar a criação de um contexto ao módulo servidor, quando é realizada a primeira requisição a uma aplicação remota;
c) encaminhar a requisição de métodos da interface remota ao módulo servidor e disponibilizar o retorno à aplicação;
d) informar o módulo servidor quando uma referência remota não é mais utilizada, para que o objeto remoto possa ser liberado da memória (DGC).

A Figura 10 apresenta as principais classes que compõem o módulo cliente.

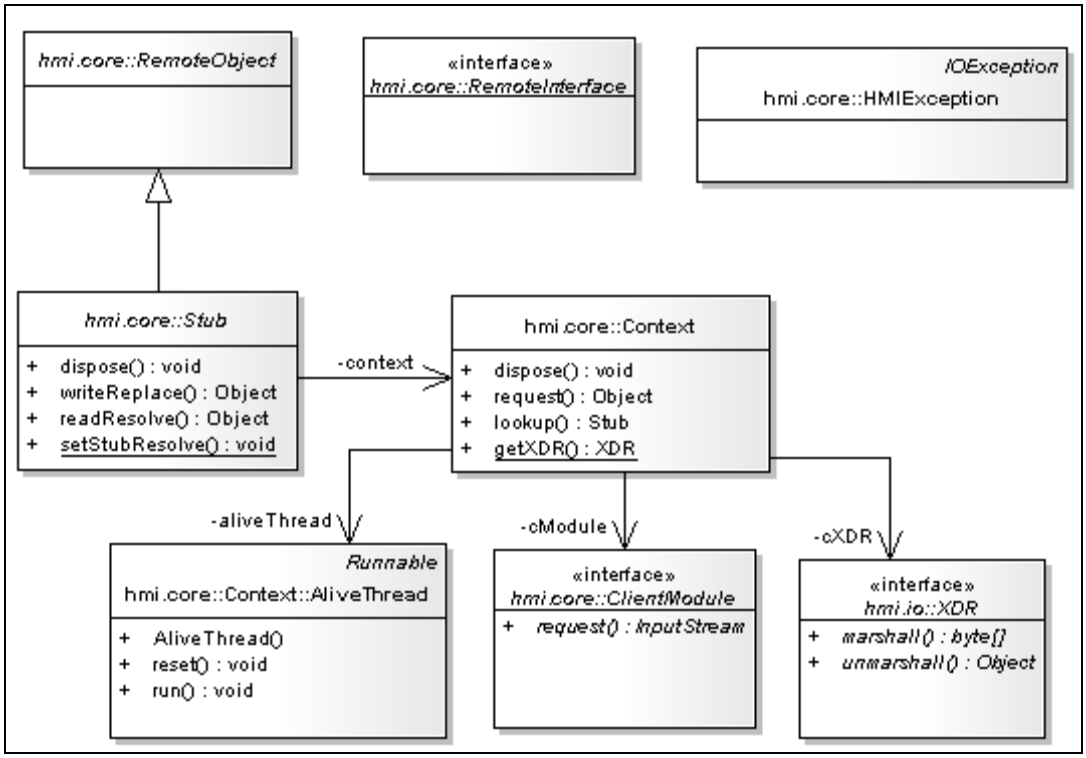


Figura 10 - Principais classes do módulo cliente

O Quadro 3 apresenta a lista de responsabilidades das classes do módulo cliente.

Classe	Responsabilidade
`hmi.core.Stub`	Classe abstrata que estende `hmi.core.RemoteObject` e é estendida pelas classes de referência remota.
`hmi.core.Context`	Armazena, no cliente, as informações do seu contexto no servidor.
`hmi.core.Context::AliveThread`	*Thread* de `hmi.core.Context`, responsável por enviar para o servidor notificações indicando que o cliente está ativo.
`hmi.core.RemoteInterface`	Identifica interfaces cujos métodos podem ser invocados remotamente. Ao implementar uma interface remota, deve-se estender esta interface.
`hmi.core.ClientModule`	Interface que deve ser implementada pelo módulo de comunicação do cliente.

Quadro 3 - Responsabilidades das classes do módulo cliente

3.2.3 Módulo de *toolkits*

Este módulo fornece a classe utilitária `HmiComplier`, aplicação responsável pela geração do s*tub e skeleton* para as classes de objetos remotos implementadas para o *middleware.*

3.2.4 Especificação das funcionalidades do *middleware*

A seguir, são detalhados os principais componentes dos módulos do *middleware*, seus relacionamentos e a dinâmica da troca de mensagens entre os componentes.

A Figura 11 mostra um diagrama de seqüência para obtenção de contexto e inicialização de aplicação.

Quando a aplicação cliente deseja obter um contexto numa aplicação remota deve invocar o método `getContext` do módulo de comunicação do cliente, passando como parâmetros o endereço ou nome do servidor, a porta de comunicação do servidor na qual a aplicação responde e o nome da aplicação.

O módulo de comunicação do cliente transforma a requisição numa sequência de bytes (XDR) e a submete via rede ao módulo de comunicação do servidor.

O módulo de comunicação do servidor, verifica se a aplicação para a qual a requisição se destina já foi iniciada. Caso não tenha sido, a mesma é instanciada é tem o seu método `start` invocado para que registre os seus objetos remotos.

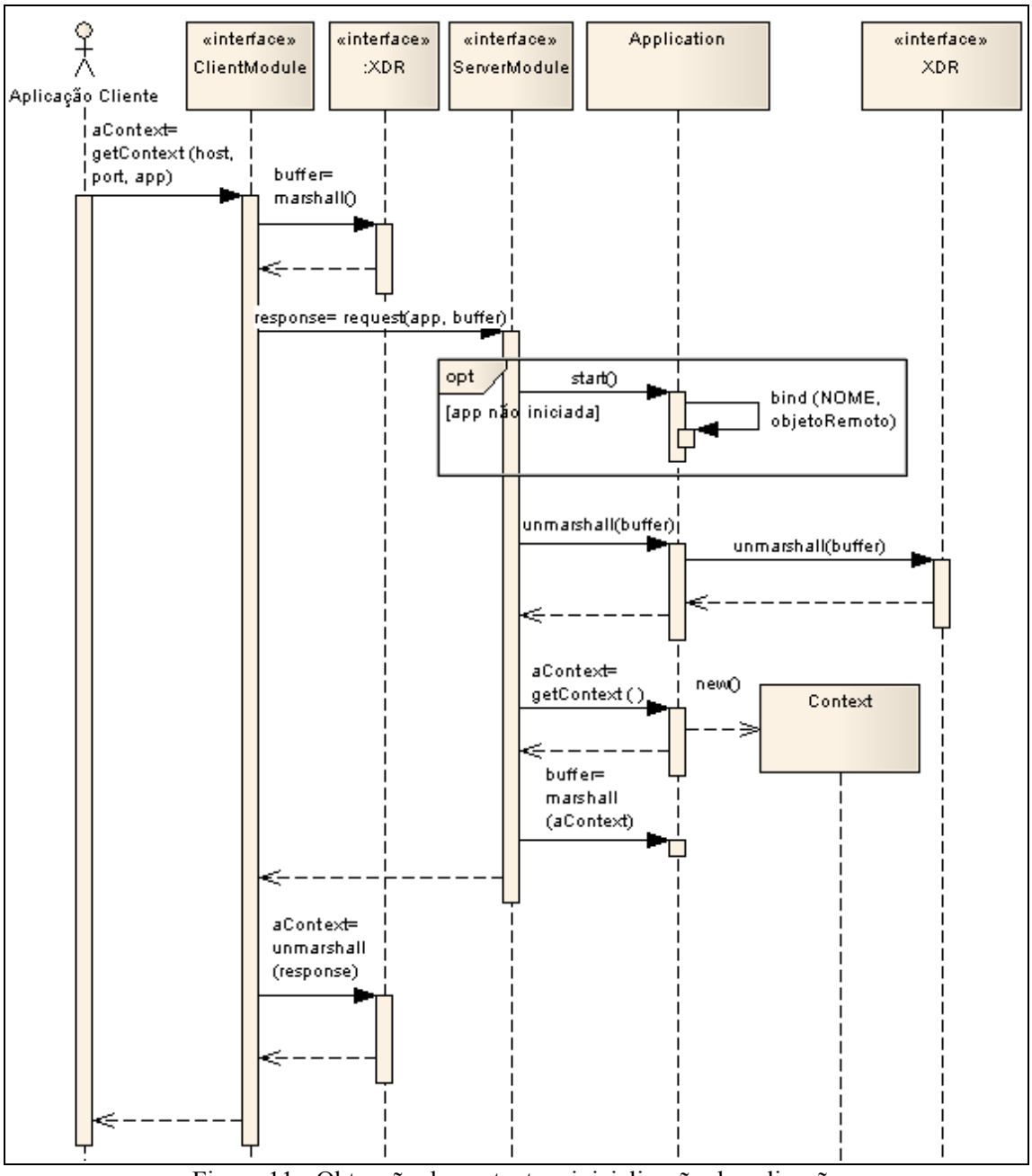


Figura 11 - Obtenção de contexto e inicialização da aplicação

Após obter a aplicação, o módulo de comunicação do servidor obtém um novo contexto através do método `getContext`, serializa-o e o devolve, via rede, para o módulo de comunicação do cliente, que o entrega para aplicação cliente que iniciou a chamada.

3.2.4.1 Interface `ServerModule`

A interface `ServerModule` determina os métodos do módulo de comunicação, apresentado no item 2.2.9, do servidor. Esta interface é implementada pela classe `ServletHMI` que estende a classe `javax.servlet.HTTPServlet`. Então, do ponto de vista do *middleware* HMI, a classe `ServletHMI` é o módulo de comunicação do servidor e é um s*ervlet*, do ponto de vista da tecnologia Java. A classe `ServletHMI` é registrada no *servlet container* para receber as invocações do protocolo HTTP.

3.2.4.2 Classe abstrata `Application`

A classe abstrata Application deve ser estendida pela aplicação que utiliza o *middleware* HMI e ter o método abstrato `start` implementado. O método `start` é chamado quando a aplicação é carregada. Utilizou-se aqui, uma abordagem semelhante ao método *`main`* das aplicações escritas em Java ou C. Considerando-se que o objetivo é a construção de aplicações baseadas em objetos distribuídos, a aplicação deve, na chamada do método `start`, registrar os objetos remotos disponibilizados por ela.

O registro de objetos remotos é realizado pela invocação do método `bind` que possui dois parâmetros: um nome (`String`) que será utilizado pelo módulo cliente para localizar o objeto remoto e a instância do objeto remoto que está sendo registrado. Os objetos registrados por `bind` ficam disponíveis durante todo o ciclo de vida da aplicação do servidor. Normalmente, uma aplicação que roda num servidor não finaliza. Assim, uma aplicação cliente que conheça o nome do objeto (primeiro parâmetro do método `bind`) e sua interface, pode invocá-lo até que o *servlet container* seja encerrado.

Para cada cliente remoto é criada uma instância de `ContextEntry`. Nela são armazenados os objetos remotos criados no contexto da aplicação cliente. Estes objetos são liberados quando a aplicação cliente é finalizada.

3.2.4.3 Interface XDR

As invocações entre o cliente e o servidor podem ter, tanto nos parâmetros quanto no retorno das chamadas, instância de objetos. Para que um objeto possa ser transmitido pela rede, é preciso transformá-lo numa seqüência de bytes. Esta transformação é conhecida como serialização ou representação externa de dados, do inglês *eXtern Data Representation* (XDR).

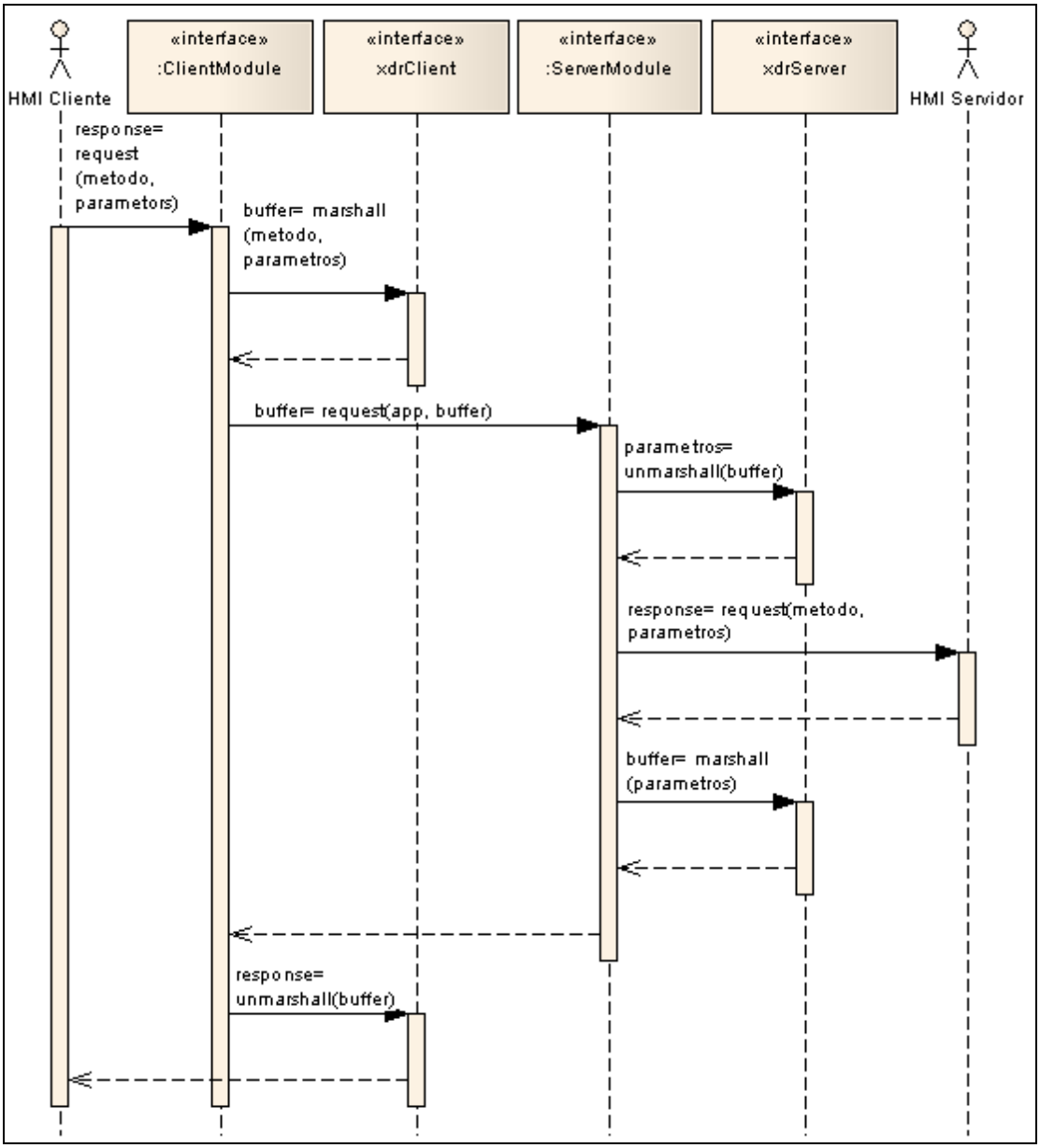


Figura 12 - Chamadas ao XDR

Dependendo da aplicação, diferentes estratégias de serialização são necessárias. Aplicações que rodam apenas sobre a plataforma JSE podem se beneficiar dos recursos de serialização fornecidos pela própria linguagem. Porém, Java ME não possui serialização nativa, ficando a cargo do programador das aplicações implementá-las. Por isso, o *middleare*

HMI utiliza a abstração XDR quando precisa serializar ou deserilizar objetos, deixando a escolha da estratégia para a classe que implementa XDR.

Para facilitar o uso do *middleware* já estão implementadas duas estratégias de XDR. A classe Java XDR utiliza o XDR nativo do Java e classe SerializavelXDR utiliza uma alternativa fornecida por HMI para serializar objetos na plataforma JME. No entanto, o programador da aplicação que utiliza HMI pode criar sua própria estratégia, implementado a interface XDR.

Por *default*, HMI utiliza a implementação JavaXDR. Para utilizar outra, deve-se especializar o método getXDR da classe `Application`.

Como mostrado na Figura 12, em cada invocação de método remoto o XDR é acionado quatro vezes. Primeiro, o módulo cliente serializa (`marshall`) a requisição para enviá-la pela rede; em seguida, o módulo servidor recebe os bytes enviados e deserializá-os (`unmarshall`) para reconstruir a requisição. Depois que a requisição é processada, o módulo servidor precisa serializar (`marshall`) o retorno para poder enviá-lo pela rede ao módulo cliente que por sua vez, recebe a seqüência de bytes e a deserealiza (`unmarshall`) para reconstruir o retorno da requisição.

Nos próximos diagramas de seqüência o XDR é omitido para economizar espaço. Porém, sempre que ocorre interação entre a aplicação e o módulo de comunicação, todas as etapas descritas no parágrafo anterior estão implícitas.

3.2.4.4 Obtenção de uma referência remota

Como apresentado na Figura 13, uma referência remota é obtida a partir de uma instância de Context. A aplicação cliente deve ter um contexto criado para a aplicação que registrou o objeto remoto e conhecer o nome com que o mesmo foi registrado.

A aplicação cliente chama o método `lookup` do objeto contexto, passando como parâmetro o nome com o qual o objeto remoto foi registrado no servidor através do método `bind`.

O objeto de contexto passa o seu identificador, o identificador do método `lookup` e o nome do objeto a ser localizado para o módulo de comunicação do cliente que invoca o XDR e repassa a seqüência de bytes, via rede, para o módulo de comunicação do servidor.

O módulo de comunicação do servidor localiza a instância da aplicação e invoca o método `request`. A aplicação, utiliza o nome do objeto para localizar o objeto remoto registrado e invoca o XDR para serializá-lo, antes de devolvê-lo para o módulo de comunicação.

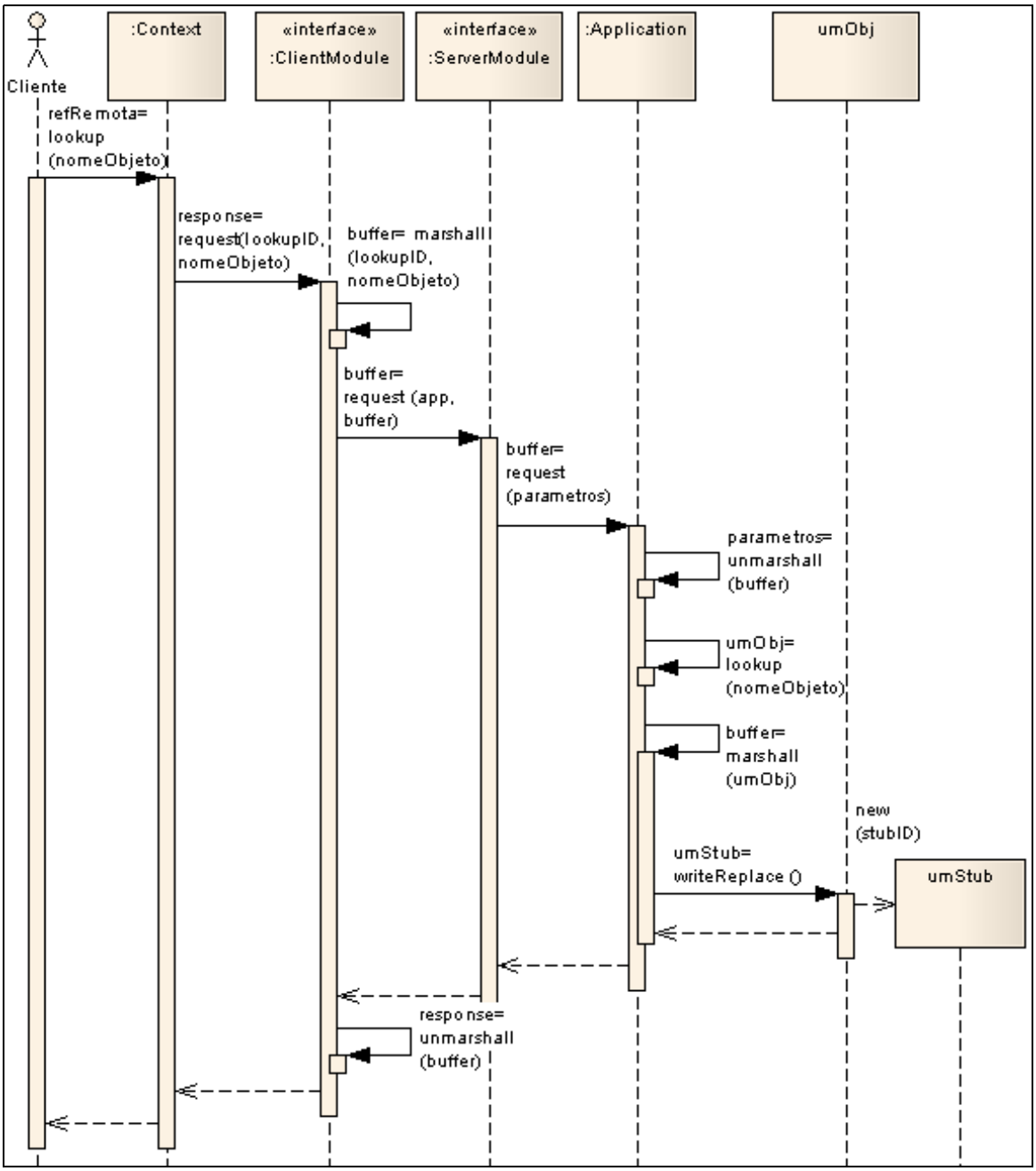


Figura 13 - Obtenção de uma referência remota

O próximo passo é um dos mais importantes do *middleware*. O módulo de comunicação não pode devolver o objeto remoto para o cliente. É necessário que uma referência remota seja criada e esta passada para o cliente. Por isso, é chamado o método `writeReplace` do objeto remoto para que a referência remota (s*tub*) seja criada. Um *stub*, possui o id do objeto remoto e uma implementação de todos os métodos da interface remota. Deve ser implementado um s*tub* para cada classe de objeto remoto.

O código da implementação do *stub* é gerado pela aplicação HmiCompiler, apresentado no item 3.2.3.

3.2.4.5 Chamada de método remoto

A Figura 14 mostra o diagrama de seqüência de uma típica chamada de método remoto.

O processo inicia-se com a chamada de um método da interface remota de um *stub* (referência remota), obtido conforme descrito no item 3.2.4.4. O *stub* implementa todos os métodos da interface remota do objeto remoto que ele referencia. Porém, não é a implementação "real". Sua função é chamar o método `request` do módulo de comunicação do cliente, passando os seguintes parâmetros:

a) `objectID`: identificador do objeto remoto. Este identificador é gerado na criação do *stub,* conforme mostrado no item 3.2.4.4;
b) `methodID`: número inteiro que identifica o método;
c) parametros: *array* com os parâmetros do método.

Em seguida, o módulo de comunicação do cliente invoca o método `marshall` do XDR para transformar a chamada acima numa seqüência de bytes e a submete com o identificador da aplicação, via rede, ao módulo de comunicação do servidor. Este por sua vez, obtém a instância de `Application` a qual a chamada se destina e submete a ela a seqüência de bytes. A aplicação, chama o método `unmarshall` do XDR para recuperar a lista de parâmetros descrita no parágrafo anterior.

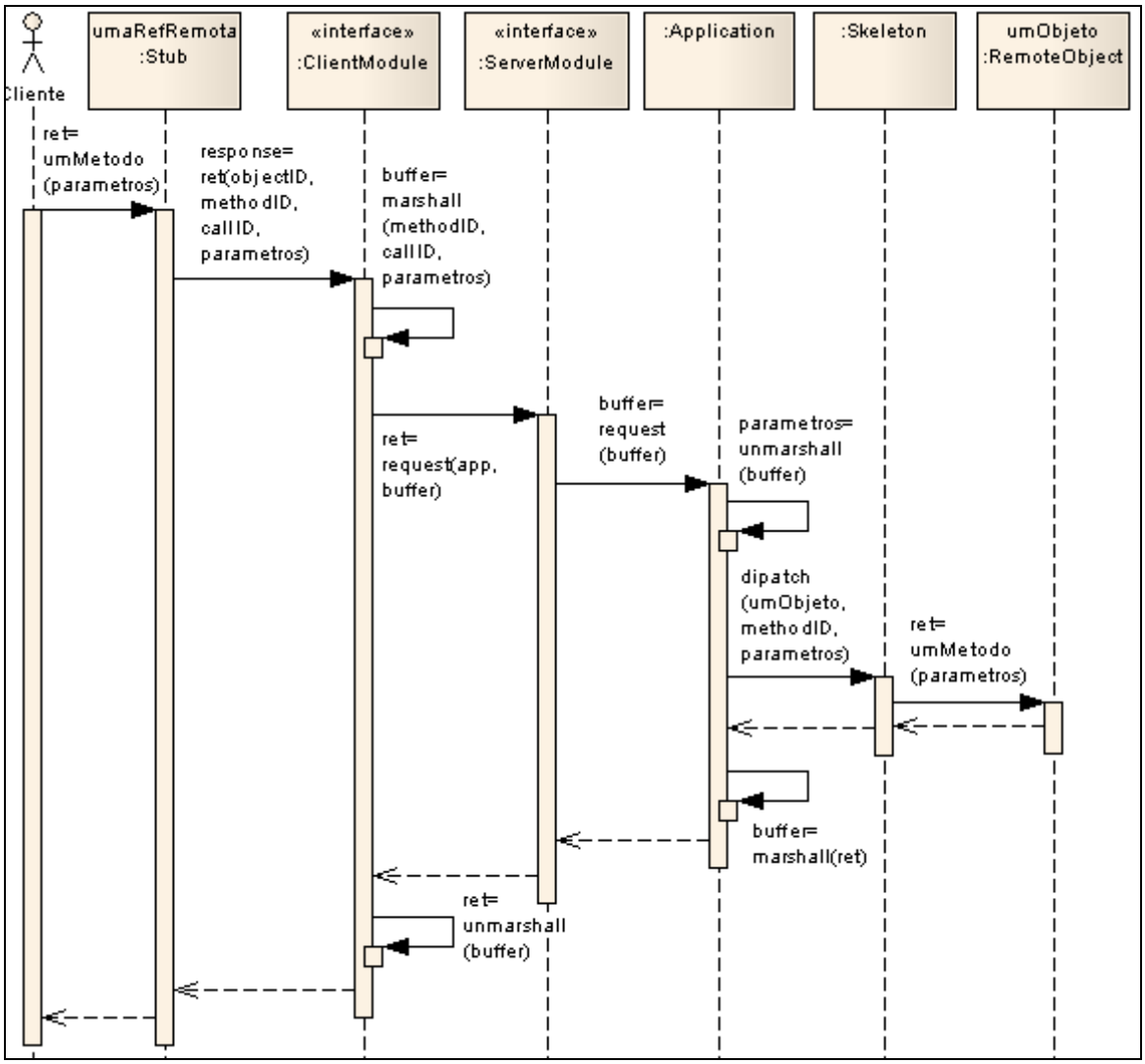


Figura 14 - Chamada de método remoto

De posse do parâmetro `objectID`, a aplicação localiza o objeto remoto no conjunto de objetos registrados. Porém, a classe `Application` não conhece a interface do objeto remoto e não tem como invocar o método desejado. Por esta razão, `Application` chama o método `dispatch` da instância de `Skeleton` correspondente à classe do objeto remoto, passando como parâmetros a instância do objeto remoto, o identificador do método que deve ser chamado e os parâmetros que devem ser passados para o método. Há uma implementação especifica de `Skeleton` para cada classe de objeto remoto. Assim, o `Skeleton` conhece a interface do objeto remoto e com base no identificado do método, chama o método desejado com os respectivos parâmetros. O código da implementação do `Skeleton` é gerado pela ferramenta HMICompiler.

3.2.4.6 Passagem de uma referência remota como parâmetro para uma chamada remota

Uma situação especial na chamada de método remoto, ocorre quando os parâmetros da chamada contêm uma referência remota (*stub*). Neste caso, o *stub* deve ser substituído pelo objeto remoto a que referencia antes de ser submetido ao `Skeleton`. Esta substituição é feita através da chamada ao método `readResolve` do *stub* pelo XDR, no momento da deserialização (*unmarshall)* dos parâmetros.

3.2.4.7 Retorno de objeto remoto

Quando uma chamada de método remoto retorna um objeto remoto, deve-se convertê-lo num referência remota (*stub*) antes de submetê-lo ao módulo de comunicação. Isto é feito através da chamada do método *`writeReplace`* da classe abstrata *`RemoteObject`* pelo XDR, no momento da serialização (*marshall)* do retorno.

3.2.4.8 Coletor de lixo distribuído

O objetivo do coletor de lixo distribuído é retirar da memória do servidor os objetos remotos, cujas referências remotas no cliente tenham sido desativadas. Em condições normais de funcionamento, o DGC trabalha em colaboração com o GC do Java. No cliente, uma referência remota é um objeto. Quando não há nenhuma referência local para ela, o GC a notifica através do método `finalize`() e em seguida a retira da memória. Quando recebe esta notificação, a referência remota envia uma requisição para o servidor, informando que uma referência remota foi desativada. O servidor decrementa o contador de referências remotas para o objeto remoto e quando este for zerado, destrói a referência local do objeto remoto para que o GC do Java o retire da memória.

Em sistemas com objetos distribuídos há necessidade de se prever a situação onde a referência remota, por falha na rede ou *crach* no cliente, não tenha chance de notificar o servidor da sua retirada da memória. O *middleware* HMI trata esta situação através da

colaboração de duas *Threads.* No cliente, um instância de `AliveThread` envia uma mensagem `isAlive` para o servidor, com um intervalo de tempo T1 configurado, indicando que o contexto está ativo. No servidor, uma instância de `HmiGC`, retira da memória os contextos que ficarem um intervalo de tempo superior a T2, sem enviarem a notificação. O *middleware* é programado para garantir que T1 seja igual a T2/2. Assim, caso o servidor espere uma notificação de continuidade a cada 60 segundos, por exemplo, o cliente enviará uma notificação a cada 30 segundos.

3.3 IMPLEMENTAÇÃO

Nessa seção são apresentados os aspectos a respeito da implementação do *middleware*, bem como as técnicas e ferramentas utilizadas. Por último, é descrita a operacionalidade da implementação através de um estudo de caso.

3.3.1 Técnicas e ferramentas utilizadas

Para a implementação do *middleware*, utilizou-se a linguagem Java e o ambiente de desenvolvimento integrado Eclipse 3.3. Do Java, utilizou-se todas as suas edições, sendo JME utilizado para desenvolvimento das classes aplicadas aos dispositivos móveis, JEE no processamento das requisições do servidor via *servlets* e JSE nas classes de uso geral. O arquivo web.xml

3.3.2 O arquivo `web.xml`

HMI implementa o módulo de comunicação do servidor como um *servlet.* A tecnologia Java exige que cada aplicação *servlet* configure o arquivo `web.xml`, informando a lista de *servlets* e suas respectivas páginas web. Para facilitar a configuração no servidor, HMI

utiliza uma abordagem que permite que o conteúdo de `web.xml`, apresentado no Quadro 4, seja o mesmo para todas aplicações, retirando do programador da aplicação a responsabilidade da configuração deste arquivo. Para isto, HMI faz todas as suas requisições para a página `hmi.jsp` que é mapeada para o módulo de comunicação do servidor (`hmi.server.http.ServletHMI`).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
<!-- Registry -->
   <servlet>
      <servlet-name>ServerModule</servlet-name>

      <servlet-class>hmi.server.http.ServletHMI</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>ServerModule</servlet-name>

      <url-pattern>/hmi.jsp</url-pattern>
   </servlet-mapping>
</web-app>
```

Quadro 4: Arquivo `web.xml` das aplicações HMI

3.3.3 O arquivo `HMI.xml`

O arquivo `HMI.xml` contém os parâmetros de inicialização da aplicação. Ele é carregado pelo módulo de comunicação do servidor quando a aplicação é acessada pela primeira vez. O Quadro 5 apresenta um arquivo `HMI.xml` típico.

```
<hmi app-class="br.com.ctx.comanda.ComandaAPP"
       timeout="600"
       xdr="br.com.ctx.io.CDCXDR"
/>
```

Quadro 5 - Arquivo `HMI.xml`

No Quadro 5, o atributo `app-class` indica a classe que deve ser chamada para iniciar a aplicação, `timeout` indica o tempo, em segundos que um contexto criado pelo cliente é mantido no servidor quando a conexão estiver inativa e `xdr` informa a classe que implementa a estratégia de representação externa de dados.

3.3.4 A inicialização da aplicação

O Quadro 6 apresenta o código fonte da incialização da apliação no servidor.

```
private synchronized static Application getApp(final File appDir)
      throws IOException, HMIException {

      final String appName = appDir.getName();

      //Busca intância da aplicação no Hash de aplicações inicalizadas
      Application result = (Application)MAP_APP.get(appName);

      //Se não localizaou,
      if (result == null) {
            //procura pelo atributo app-class no hmi.xml
            String appClass = xml.findItemAttribute("//hmi", "app-class");

            • • •
            //obtém uma instância da aplicação
            result= (Application)Class.forName(appClass).newInstance();

            //executa o método start
            result.start(appName, xml);

            // e armazena no hash de aplicações inicalizadas
            MAP_APP.put(appName, result);

            • • •
      }
      return result;
}
```

Quadro 6 - Inicialização da aplicação

Sempre que o módulo de comunicação do servidor recebe uma solicitação, invoca o método `getApp` para obter a instância da aplicação a qual a chamada se destina. Caso a aplicação ainda não tenha sido instanciada, uma instância é criada e o método `start` invocado para que os objetos remotos da aplicação sejam registrados.

3.3.5 Registro de objeto remoto

O registro do objeto remoto é realizado pelo método `bind` da classe abstrata `Application`. Como mostrado no Quadro 7.

```
public void bind(String name, RemoteObject remote) throws HMIException {
      MAP_REGISTRY.put(name, remote);
}
```

Quadro 7 - Fonte do método de registro de objetos remotos

Observa-se que o método `bind` é bastante simples. Apenas registra o objeto remoto numa estrutura `hash` utilizando o nome como chave.

3.3.6 Processamento das requisições

De posse da instância da aplicação, o módulo de comunicação chama o método `request` passando o `stream` da requisição como parâmetro. O método `request` invoca o método `getRequestParams` que é o responsável por solicitar ao XDR que realize o `unmarshall` do `stream`.

As requisições do *middleware* possuem o formato apresentado no Quadro 8:

```
GET_NEW_CONTEXT | LOOKUP contextID nome | CALL objectID methodID parametros | DISPOSE_OBJECT objectID | ALIVE contextID | DISPOSE_CONTEXT contextID
```

Quadro 8 - Formato das requisições do middleware

O Quadro 9 apresenta parte do código do método `request` da classe `Application`. O primeiro parâmetro do `request` indica a ação a ser realizada. Basicamente, o método `request` identifica a ação que a requisição está invocando e chama o método responsável por tratá-la. Quanto o método que trata a ação lança uma exceção, esta exceção torna-se o retorno do `request`. Quando o módulo de comunicação do cliente recebe uma exceção como retorno, este a lança no contexto do cliente, dando à aplicação a oportunidade de tratá-la.

No momento da serialização (*marshall)* do retorno, uma *abstract factory*[9] de *`Stubs`* é atribuída à classe `RemoteObject`. Caso um objeto do tipo `RemoteObject` seja serializado, este será submetido à *factory* para ser substituído pela referência remota (*stub)* correspondente. O código precisa ser protegido porque duas aplicações diferentes não devem serializar objetos simultaneamente pois podem ter diferentes *factories* de *stubs*.

[9] Padrão de projeto utilizado para criar instâncias de diferentes classes que implementam uma interface pública comum (GAMMA, Erich *et al*. 2000, p. 95).

```
public byte[] request(InputStream request) throws IOException {
      Object result = null;
      int context = 0;

      try {
            final Object[] params = getRequestParams(request);

            // params[0] = Ação do Registry
            final byte action = ((Byte) params[0]).byteValue();
            switch (action) {

            • • •
                  case Stub.CALL :
                  context = getContextID(((Integer)
params[1]).intValue());

                  int objectID = ((Integer) params[1]).intValue();
                  int methodID = ((Integer) params[2]).intValue();
                  Object parametros = params[3];

                  result = call(objectID, methodID, parametros);
                  break;

            • • •
            } catch (Throwable t) {
                  result = t;
            }

            synchronized (StubResolveMutex.getInstance()) {
                  currentContext = context;
                  RemoteObject.setStubFactory(getFactory());
                  byte[] ret = getXDR().marshall(result);
                  RemoteObject.setStubFactory(null);
                  return ret;
            }

      }
```

Quadro 9 - Fonte do método *request* da classe *Application*

Em todas requisições o primeiro parâmetro (`((Byte) params[0]).byteValue()`) indica o tipo de chamada. No exemplo do Quadro 9 trata-se de uma invocação de método. Neste caso, o segundo parâmetro é o identificador do objeto, o terceiro o identificador do método e o quarto os parâmetros que devem ser submetidos ao método.

3.3.7 Implementações de XDR

O *middleware* possui duas implementações de XDR. Aplicações que rodam na plataforma JSE ou JEE utilizam a implementação `JavaXDR`, que tem seu código apresentado

no Quadro 10.

Como JSE implementa *marshall* de objetos com a classe `java.io.ObjectOutputStream` e *unmarshall* na classe `java.io.ObjectInputStream`, o *middleware* faz uso destas implementações já disponibilizadas.

Aplicações que rodam no ambiente JME, utilizam a implementação `SerializavelXDR`, apresentada no Quadro 11.

```
public class JavaXDR implements XDR {

   public byte[] marshall(Object object) throws IOException {
       final ByteArrayOutputStream bos = new ByteArrayOutputStream();
       final ObjectOutputStream out = new ReplaceObjectOutputStream(bos);
       out.writeObject(object);
       out.flush();
       return bos.toByteArray();
   }

   public Object unmarshall(InputStream is) throws IOException, ClassNotFoundException
{
       final ObjectInputStream in = new ObjectInputStream(is);
       Object ret = in.readObject();
       if (ret instanceof ExceptionWrapper) {
              ExceptionWrapper ew = (ExceptionWrapper)ret;
              return Reflector.makeNew(ew.getException(),ew.getMessage());
       }
       return ret;
   }


}
```

Quadro 10 - Código fonte de JavaXDR

`JavaXDR` utiliza os recursos de serialização nativos do Java. No Quadro 11 é apresentado o código fonte de `SerializavelXDR` uma alternativa do *middleware* para serialização de objetos.

```
public class SerializavelXDR implements XDR {

       public byte[] marshall(Object object) throws IOException {
          return SerializadorSE.getInstance().serializa(object);
       }

       public Object unmarshall(InputStream is) throws IOException,
                     ClassNotFoundException {
          return SerializadorSE.getInstance().restaura(new DataInputStream(is));
       }
}
```

Quadro 11 - Código fonte de XDR para JME

A diferença entre `JavaXDR` e `SerializavelXDR` é que o último utiliza a classe `hmi.Serializdor` para serializar e restaurar objetos. Serializador é uma classe do *middleware* que tem como objetivo fornecer uma alternativa de serialização de objetos no ambiente JME, que não a fornece de forma nativa. O Quadro 12 apresenta um fragmento do código da classe `hmi.Serializador`.

```
public byte[] serializa(final Object o, ClassMapper map) throws IOException {
        final ByteArrayOutputStream bout = new ByteArrayOutputStream();
        final DataOutputStream dout = new DataOutputStream(bout);
        if (o == null) {
                dout.writeByte(TYPE_NULL);
        } else if (o instanceof Byte) {
                dout.writeByte(TYPE_BYTE);
                dout.writeByte(((Byte) o).byteValue());
        } else if (o instanceof Short) {
                dout.writeByte(TYPE_SHORT);
                dout.writeShort(((Short) o).shortValue());


         • • •


        } else if (o instanceof Serializavel) {
                dout.writeByte(TYPE_SERIALIZAVEL);
                String nome = null;
                if (map != null) {
                        int tipo = map.getDataType(o.getClass());
                        nome = o.getClass().getName();
                        dout.writeInt(tipo);
                } else {
                        nome = o.getClass().getName();
                        dout.writeUTF(nome);
                }
                ((Serializavel) o).serializa(dout);
        }
        else if (o instanceof RemoteObject) {
                dout.write(serializa (((RemoteObject)o).writeReplace()));

        } else if (o.getClass().getName().charAt(0) == '[') {
                serializaArray(o, dout, map);
        dout.flush();
        return bout.toByteArray();
}
```

Quadro 12 - Código fonte da classe `hmi.Serializador`

A classe `hmi.Serializador` utiliza a classe Java `java.io.DataOutputStream` para serializar cada atributo (tipo primitivo ou String) separadamente, suprindo a ausência da classe `java.io.ObjectOutputStream` na plataforma JME.

3.3.8 Chamada de método remoto

A chamada de método remoto é executada com a colaboração de várias classes. A chamada inicia com o acionamento de um método da interface remota, implementado pelo *stub* (referência remota). O Quadro 13 mostra uma implementação típica de um método da interface remota no *stub*.

```
public final class Session_HmiStub extends Stub implements ISession {

      • • •

  // ID -1365533222 "login"
  public void login(String usuario, String senha) throws IOException, CTxException {
            try{
                  this.call( -1365533222, new java.lang.Object[]{usuario,senha});
            }
            catch (Throwable t){
                  if (t instanceof java.io.IOException){
                        throw (java.io.IOException)t;
                  }
                  if (t instanceof br.com.ctx.CTxException){
                        throw (br.com.ctx.CTxException)t;
                  }
                  throw new br.com.ctx.hmi.HMIException(t);
            }
      }

      • • •
```

Quadro 13 - Implementação de um método do `Stub`

No exemplo do Quadro 13, a classe `Session_HmiStub` implementa a interface `ISession` que define a interface remota da classe `Session`. O método `login` é um dos métodos da interface remota. A implementação delega a chamada ao método `call` da classe `Stub` apresentado no Quadro 14, passando como parâmetros o identificador do método e um `array` de objetos com os valores dos parâmetros. Quando o método não possui parâmetros é passado `null` para este array.

```
      public Object call(int metodo, Object parametros) throws Throwable {
                  final Object[] params = new Object[]{
                        new Byte(Stub.CALL),
                        new Integer(iObjectId),
                        new Integer(metodo),
                        parametros};
            Context c = Context.getObjectContext(iObjectId);
            return c.request(params);

      }
```

Quadro 14: Implementação do método `call` do `Stub`

O método *`call`* obtém a instância do contexto corrente, da qual invoca o método `request`, apresentado no Quadro 15, passando como parâmetro um *`array`* de objetos contendo a identificação da ação (*`CALL`*), o identificador do objeto remoto, o identificador do método e o *`array`* de parâmetros.

```
public Object request (Object param) throws  Throwable {
      InputStream is;
      try {
            //obtem o array de bytes (marshall)
            byte[] buffer = getXDR().marshall(param);

            //submete ao módulo de comunicação
            is = getModule().request(iServiceUrl, iPort, iApp, buffer);

            //obtém o objeto de retorno (unmarshall)
            Object r = getXDR().unmarshall(is);

            is.close();
            if (r instanceof Throwable) {
                  throw (Throwable)r;
            }
            return r;
      } catch (Throwable e) {
            if (e instanceof HMIException) {
                  throw (HMIException) e;
            }
            e.printStackTrace();
            throw new HMIException(e);
      }
}
```

Quadro 15 - Fonte do método *request* da classe `Context`

O método `request` do contexto invoca o XDR para serializar os parâmetros e em seguida, submete o buffer ao módulo de comunicação do cliente (Quadro 16), através do método `request`, passando além do buffer, o endereço do servidor, a porta de comunicação, e aplicação onde o contexto está instanciado. O método `request` retorna um *buffer* com o retorno da chamada do método remoto serializado. O buffer é submetido ao XDR para recuperar o retorno.

```
public class HTTPClientModule implements ClientModule {

	public InputStream request(final String server,
			int port,
			String app,
			final byte[] stream) throws Throwable {

		// Compacta o stream
		ByteArrayOutputStream bos = new ByteArrayOutputStream();
		final DeflaterOutputStream compressor = new DeflaterOutputStream(bos);
		compressor.write(stream);
		compressor.finish();
		bos.flush();
		byte[] buffer = bos.toByteArray();

		// obtém a URL
		String host = "http://" + server + ":" + port + "/" + app +
"/hmi.jsp";
		final URL url = new URL(host);

		// Monta o Post
		final HttpPostRequest http = new HttpPostRequest();
		http.setParameter("param", buffer);
		byte[] b = http.post();

		// submete o post
		final HttpURLConnection conn = (HttpURLConnection)
url.openConnection();
		conn.setDoOutput(true);
		conn.setDoInput(true);
		conn.setRequestMethod("POST");
		conn.setRequestProperty("Content-Type", http.getContentType());
		final OutputStream os = conn.getOutputStream();
		os.write(b);
		os.flush();
		os.close();

		// devolve o Stream de retorno do POST
		return new InflaterInputStream(conn.getInputStream());

	}
}
```

Quadro 16 - Módulo de comunicação do cliente

O código do Quadro 16 é o limite do *middleware* no lado do cliente. A partir deste ponto, a comunicação fica à cargo do protocolo de comunicação, neste caso, o HTTP até o acionamento do módulo de comunicação do servidor, mostrado no Quadro 17. O módulo de comunicação do servidor implementado pelo *middleware* é um *servlet*. Do ponto de vista da tecnologia Java, um *servlet* deve estender a classe `HttpServlet` e sobrescrever o método `doPost`. Quando uma requisição HTTP é direcionada à URL onde o *servlet* está registrado, o método `doPost` é invocado. No caso do *middleware*, o servlet (módulo de comunicação do servidor) é invocado quando o código do Quadro 16 é executado no cliente.

```
public final class ServletHMI extends HttpServlet implements ServerModule{

       • • •
  protected void doPost(final HttpServletRequest request,
       final HttpServletResponse response) throws ServletException,
                    IOException {
       Application app = null;
       final File webAppDir = new File(getServletContext().getRealPath(""));
       app = getApp(webAppDir);

       if (app != null) {
              byte[] buffer = app.request(getRequest(request));
                    System.out.println("Enviados "
                                 + buffer.length
                                 + "bytes para "
                                 + request.getRemoteAddr());
                    sendResponse(response, buffer);
              }
       }


       • • •
}
```

Quadro 17 - Código fonte do módulo servidor

O módulo de comunicação do servidor identifica a aplicação a qual a requisição se destina e invoca o seu método `request`, apresentado no Quadro 18. Em seguida, envia o retorno do método `request` ao módulo de comunicação do cliente.

```
public abstract class Application implements StubResolve {
       ...
 public byte[] request(InputStream request) throws IOException {
       Object result = null;
       int context = 0;

       try {
              final Object[] params = getRequestParams(request);
              // params[0] = Ação do Registry
              final byte action = ((Byte) params[0]).byteValue();
              switch (action) {
       ...
                 case Stub.CALL :
                     context = getContextID(((Integer) params[1]).intValue());
                     int objectID = ((Integer) params[1]).intValue();
                     int methodID = ((Integer) params[2]).intValue();
                     Object parametros = params[3];
                     result = call(objectID, methodID, parametros);
                     break;
       ...

              synchronized (StubResolveMutex.getInstance()) {
                     currentContext = context;
                     RemoteObject.setStubFactory(getFactory());
                     byte[] ret = getXDR().marshall(result);
                     RemoteObject.setStubFactory(null);
                     return ret;
              }

       }
```

Quadro 18 - Método `request` da classe `Application`

O método `request` da classe `Application` chama o método `getRequestParams`, apresentado no Quadro 19.

```
private Object[] getRequestParams(InputStream is) throws IOException,
            ClassNotFoundException {
      synchronized (StubResolveMutex.getInstance()) {
            Stub.setStubResolve(this);
            return (Object[]) getXDR().unmarshall(is);
      }
}
```

Quadro 19 - Método `getRequestParams` da classe `Application`

O método `getRequestParams` tem por objetivo ativar o XDR para transformar o *stream* de bytes recebido do protocolo de comunicação (HTTP) na lista de parâmetros enviados para a aplicação (*unmarshall)*. Após a execução deste método, tem-se recuperado o *array* de objetos enviado pelo método *call* da classe S*tub*, apresentado no Quadro 14.

Caso um dos objetos recuperados pelo XDR seja uma referência remota, esta deve ser substituída pelo objeto remoto correspondente. A substituição é feita com colaboração de XDR, `Stub` e `Application`. O primeiro passo é feito com a injeção de dependência de `Application` sobre o `Stub` através do método `setStubRersolve`.

Quando o XDR identifica que uma instância de `Stub` foi deserealizada, invoca o seu método `readResolve` mostrado no Quadro 20, que por sua vez, chama o método `getRemoteObject` da instância de `Application` injetada no método `getRequestParams` de `Application`.

```
public abstract class Stub extends RemoteObject implements Serializavel{
      ...
      public Object readResolve() {
            return app!=null?app.getRemoteObject(getObjectId()):this;
      }
      ...
      public static void setStubResolve(StubResolve stubResolve) {
            Stub.app = stubResolve;

      }
      ...
}
```

Quadro 20 - Os métodos `setStubResolve` e `readResolve`

Após obter os parâmetros, o método `request` de `Application` delega a chamada ao método `call`, apresentado no Quadro 21, informando o identificador do objeto, identificador do método e lista de parâmetros.

```
private Object call(final int objectID,
            final int methodID,
            final Object parametros) throws HMIException, IOException {

      // Obtem o contexto do objeto
      ContextEntry ce = getContextEntry(getContextID(objectID));

      // Contexto não é cadidato à exclusao do DGC por inatividade
      ce.isAlive();

      // Obtem o objeto remoto ao qual a chamada se destina
      RemoteObject ro = ce.getRemoteObject(objectID);

      // Obtem a instancia de Skeletton correspondente ao objeto remoto
      Skeleton skeleton = getSkeleton(ro);

      // submete a chamada ao objeto remoto
      return skeleton.dispatch(ro, methodID, parametros);
}
private Skeleton getSkeleton(RemoteObject ro) throws HMIException {
      Skeleton result = (Skeleton) SKELETONS.get(ro.getClass());
      if (result == null) {
            result = (Skeleton) getFactory().makeSkeleton(ro);
            SKELETONS.put(ro.getClass(), result);
      }
      return result;
}
```

Quadro 21 - Métodos `call` e `getSkeleton` da classe `Application`

A implementação do `Skeleton` é gerada pela ferramenta HmiCompiler, especificada no tópico 3.2.4.5. Para cada classe de objeto remoto há uma classe de `Skeleton` correspondente. No Quadro 22 é apresentado um trecho do código de um `Skeleton`.

```
public final class Session_HmiSkel extends br.com.ctx.hmi.Skeleton {

      // ID -1365533222 "login"
  private void logout(Session ro,String usuario, String senha) throws IOException,
      CTxException {
            ro.logout(usuario,senha);
      }

      // implementação do método "dispatch"
 public Object dispatch(RemoteObject ro, int methodId, Object arg) throws
      HMIException {
      java.lang.Object[] args = null;

    switch (methodId) {
      case -1365533222 : // ID -1365533222 "login"
         args = (java.lang.Object[]) arg;
         this.login((br.com.ctx.sessions.Session)ro,(String) args[0],(String)
            args[1]);
         break;
    }
}
```

Quadro 22 - Código fonte de um `Skeleton`

O `Skeleton` gerado a partir da classe do objeto remoto, conhece sua interface pública e pode chamar seus métodos.

3.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO

O teste da operacionalidade do *middleware* foi realizado sobre a aplicação Comanda Eletrônica (CE), voltada à automação do atendimento de restaurantes. A especificação da aplicação foi construída no EA. A implementação foi escrita em Java. Para rodar a camada cliente, utiliza-se a configuração CDC do JME e no servidor, a aplicação utiliza o *Tomcat* como *container*.

3.4.1 Requisitos da Comanda Eletrônica

A CE deve rodar em dispositivos móveis e seus usuários são os garçons do restaurante. Quando um garçom realiza um atendimento, informa o número da mesa e os itens do cardápio que os clientes desejam com suas respectivas quantidades.

Enquanto ocupar uma mesa, o mesmo grupo de clientes pode realizar diversos pedidos com diferentes garçons. O garçom que realiza um atendimento à mesa, deve ser capaz de consultar os pedidos realizados anteriormente ao mesmo grupo, mesmo que realizados por outros garçons.

Os ocupantes de uma mesa podem solicitar o fechamento da conta a qualquer garçom do restaurante. Após o fechamento da mesa, a CE não deve permitir que novos pedidos sejam cadastrados, até que o sistema legado processe a conta do grupo de pessoas que está saindo.

Os itens do cardápio são organizados em famílias. O garçom deve poder consultar o cardápio e os itens devem estar agrupados por família. Para cada família de itens há um conjunto pré-definido de observações que o garçom pode aplicar ao item no momento do pedido.

É requisito da CE a integração em tempo real com o sistema legado do restaurante. Devido a este requisito e a limitada capacidade de processamento dos dispositivos móveis, optou-se por uma arquitetura em camadas. Deseja-se que a interface com o usuário rode no dispositivo móvel e esta interaja com um servidor de aplicação através de objetos distribuídos.

A aplicação CE foi construída, seguindo a prática de projeto orientado pelo modelo de domínio, do inglês *Domain Driven Design* (DDD). Segundo Richardson (2007, p. 66),

embora os modelos de domínios de diferentes problemas sejam bastante diferentes, em DDD, as classes de uma aplicação podem ser categorizadas, segundo o seu papel, em uma das seguintes categorias:

a) Entidades: representam uma identidade de negócio distinta que é separada dos valores de seus atributos. Duas entidades são diferentes mesmo se os valores de seus atributos forem os mesmos e não podem ser utilizadas intercambiavelmente. Frequentemente elas correspondem a conceitos do mundo real que são centrais para o modelo de domínio;
b) v*alue Objects:* são objetos definidos primariamente pelos valores de seus atributos;
c) Fábricas: são classes que definem métodos para criar instâncias de outras classes;
d) Repositórios: gerenciam coleções de entidades e definem métodos para encontrar e remover entidades;
e) Serviços: implementam o fluxo de trabalho da aplicação. Essas classes são a força diretriz da aplicação e contêm os métodos que realizam um caso de uso. Geralmente os serviços incluem comportamentos que não podem ser atribuídos a uma única entidade e são constituídos por métodos que agem sobre diversos objetos.

A Figura 15 apresenta os casos de uso atendidos pela Comanda Eletrônica. Segundo a filosofia DDD, cada caso de uso dará origem a uma classe de serviço.

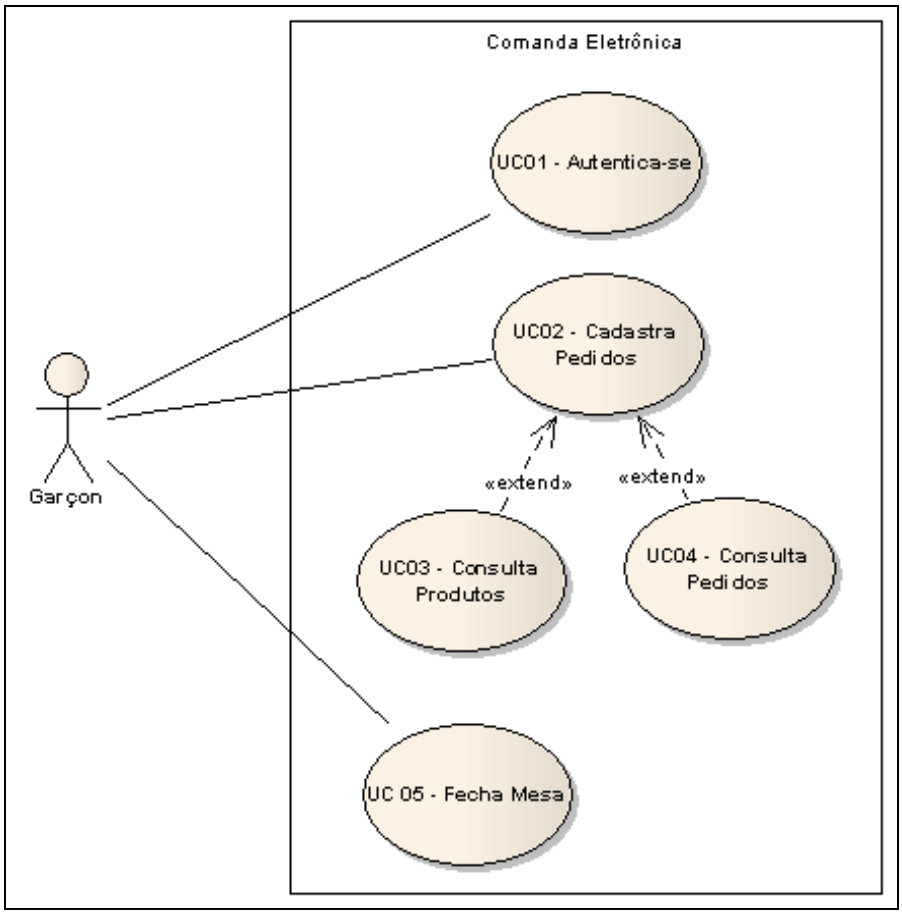


Figura 15 - Casos de Uso da Comanda Eletrônica

O protótipo da tela de cadastro de pedidos é mostrado na Figura 16.

Figura 16 - Tela de cadastro de pedidos

O garçom inicia o atendimento informando o número da mesa. Em seguida, para cada item, informa o código, a quantidade desejada, a posição na mesa da pessoa que pediu e uma observação pré-cadastrada. Caso o garçom não saiba o código do item, acessa a lista de itens do cardápio através do ícone com formato de lupa.

3.4.2 Modelo de domínio da Comanda Eletrônica

A partir dos requisitos da CE, chegou-se ao modelo das entidades do domínio apresentado na Figura 17.

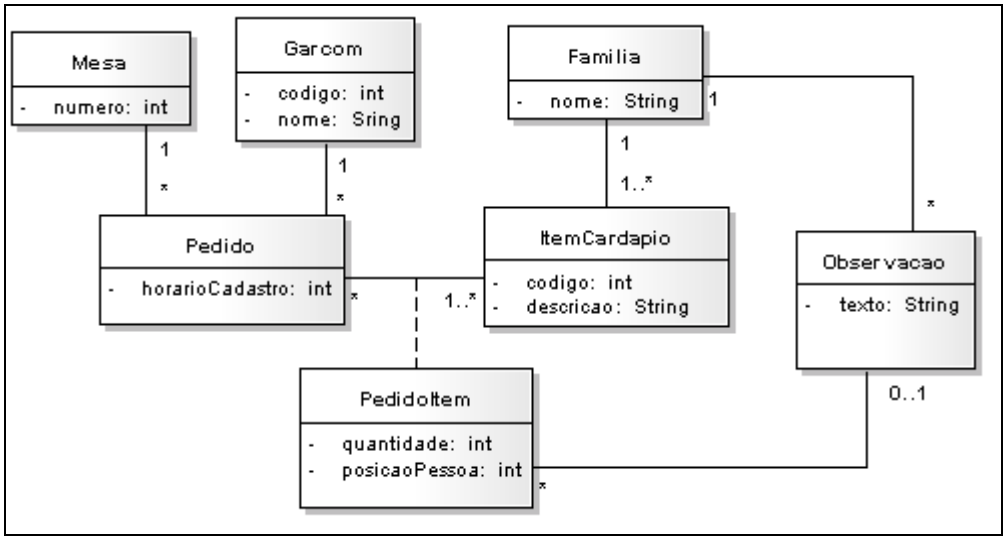


Figura 17 - Entidades do modelo de domínio da CE

Das classes do modelo da Figura 17, apenas `Pedido` e `PedidoItem` são instanciadas pela aplicação CE. As demais tem origem na sistema legado.

Os casos de uso, apresentados na Figura 15, dão origem aos serviços e repositórios apresentados na Figura 18.

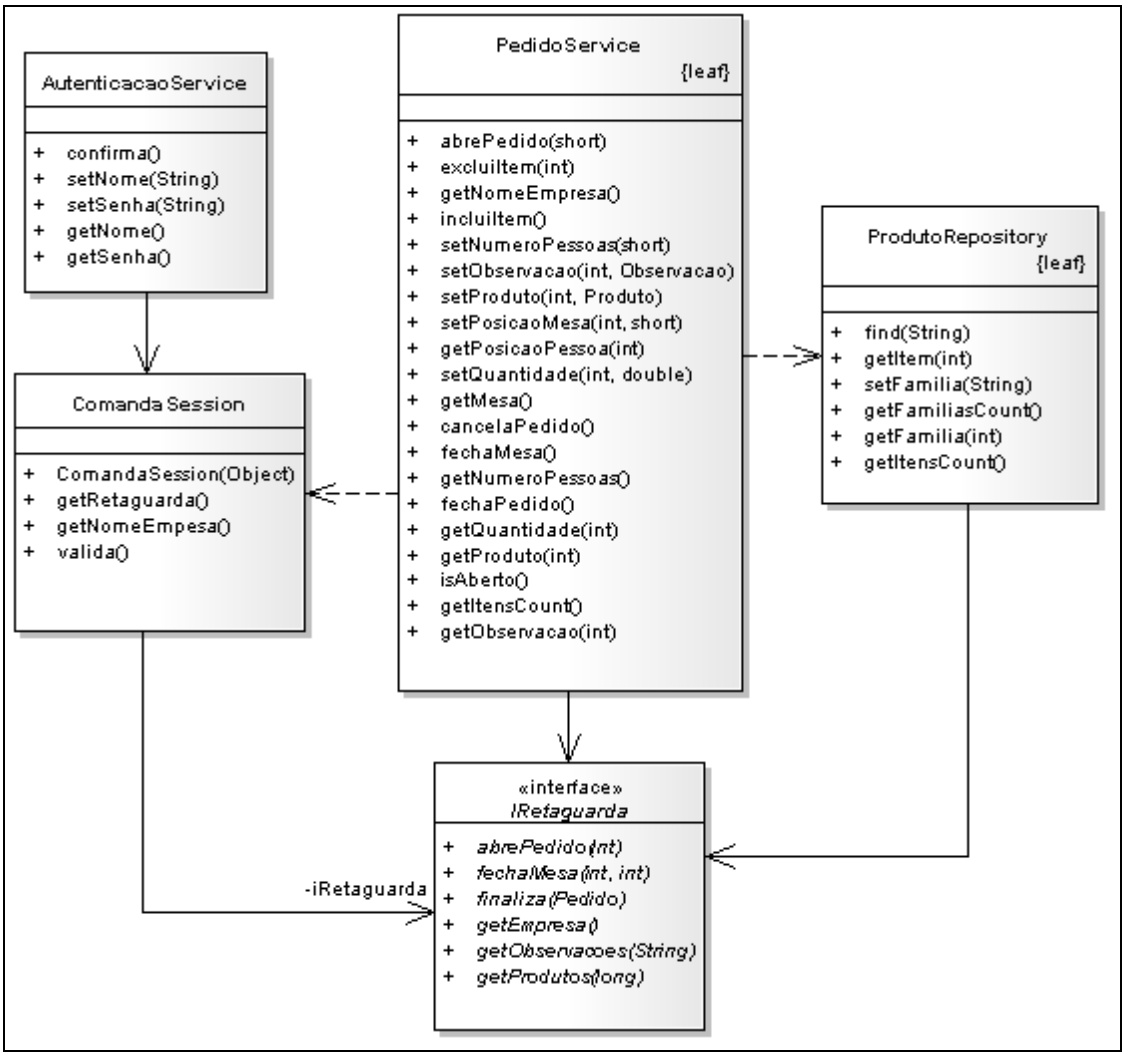


Figura 18 - Serviços do modelo de domínio da CE

Cada serviço apresentado na Figura 18 realiza um caso de uso. Todos os serviços dependem do fornecido pelo sistema legado. O serviço de pedido depende do repositório de produtos para possibilitar a localização de um item do cardápio e do repositório de pedidos para apresentar o histórico de pedidos da mesa.

O modelo da Figura 18 funciona bem quando o modelo de domínio roda inteiramente numa única máquina. É o caso das aplicações *desktop* e de muitos dos sistemas *web*. Porém na aplicação CE, adotou-se uma abordagem diferente. Para maximizar a produtividade a aplicação cliente não roda num browser. Implementou-se uma arquitetura distribuída onde uma aplicação Java é instalada no dispositivo móvel, executando a interface com o usuário no próprio dispositivo. Além disso a porção dos serviços que não depende de informações do servidor também e executada no próprio dispositivo móvel, minimizando requisições à rede. A Figura 19 apresenta o diagrama de pacotes deste modelo.

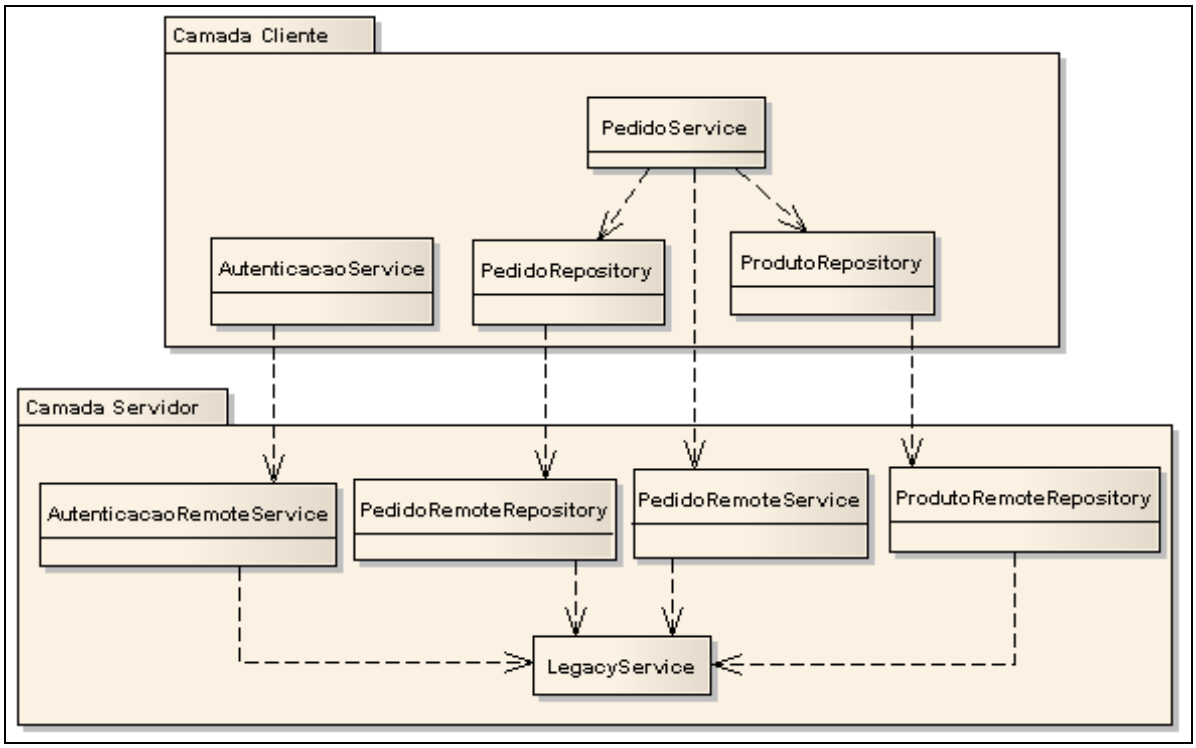


Figura 19 - Serviços do modelo de domínio distribuído

A arquitetura da Figura 19 divide os serviços do modelo de domínio em duas partes. A primeira roda no dispositivo móvel e resolve localmente o que pode ser resolvido independente do servidor. A segunda, é composta por objetos remotos, rodando no servidor de aplicação é chamada pela primeira, para tratar as invocações que não podem ser resolvidas localmente.

O diagrama de seqüência da Figura 20 mostra a colaboração entre os serviços para resolver o cenário principal do caso de uso cadastra pedido. No primeiro passo, o garçom solicita que um novo pedido seja criado, informando o número da mesa. Esta requisição é passada para o serviço remoto para verificar se a mesa está disponível para atendimento (não possui fechamento pendente). Em seguida, o garçom informa o código de um item do cardápio. Novamente a requisição precisa ser submetida ao servidor remoto para que o item seja localizado.

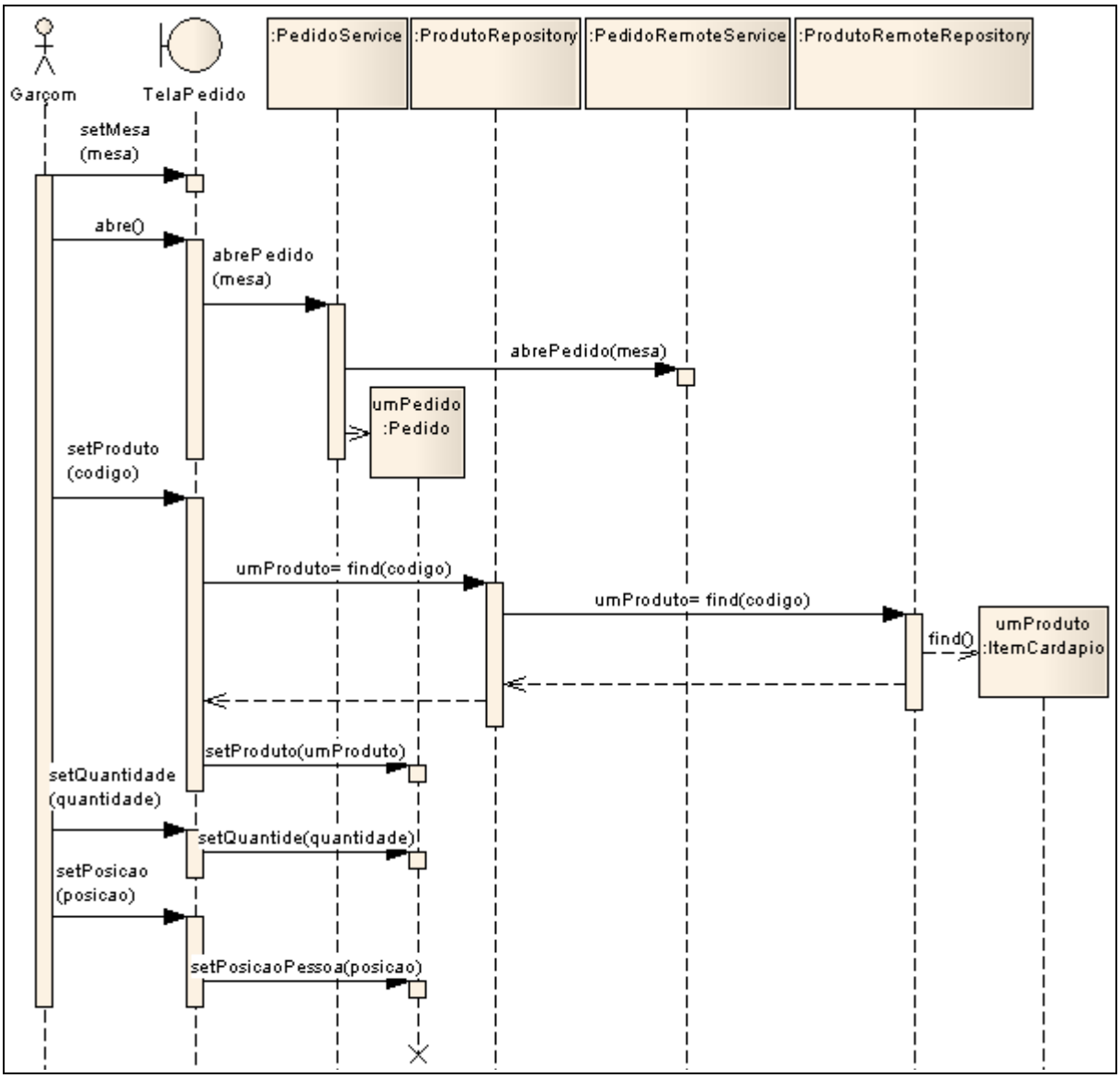


Figura 20 - Seqüência da inclusão de um item no pedido

Nas chamadas seguintes, `setProduto`, `setQuantidade` e `setPosicaoPessoa` não precisam de processamento no servidor e são tratadas na camada cliente, evitando chamada à rede.

Apresentada a arquitetura de aplicação CE, precisa-se tratar a chamada aos métodos dos objetos remotos. Um objeto que está rodando na camada cliente não tem como invocar um método de um objeto instanciado na camada do servidor, utilizando apenas os recursos nativos da linguagens de programação orientadas a objetos. É neste ponto que o *middleware* entra em ação. Seu papel é permitir que objetos distribuídos nos diferentes nós que compõem o sistema invoquem métodos uns dos outros com o mínimo de impacto na sintaxe e semântica da linguagem de programação que hospeda o *middleware*, Java, neste exemplo.

3.4.3 Implementação de classes de objeto remoto

Nos parágrafos a seguir são apresentadas as implementações de algumas classes de objetos remotos da aplicação CE.

Durante a execução de CE objetos instanciados em nós clientes, precisam solicitar que instâncias de serviços remotos sejam iniciadas no servidor. Para isto, utiliza-se uma fábrica de objetos. A Figura 21 apresenta a classe de objeto remoto `ComandaRemoteFactory` e alguns de seus relacionamentos.

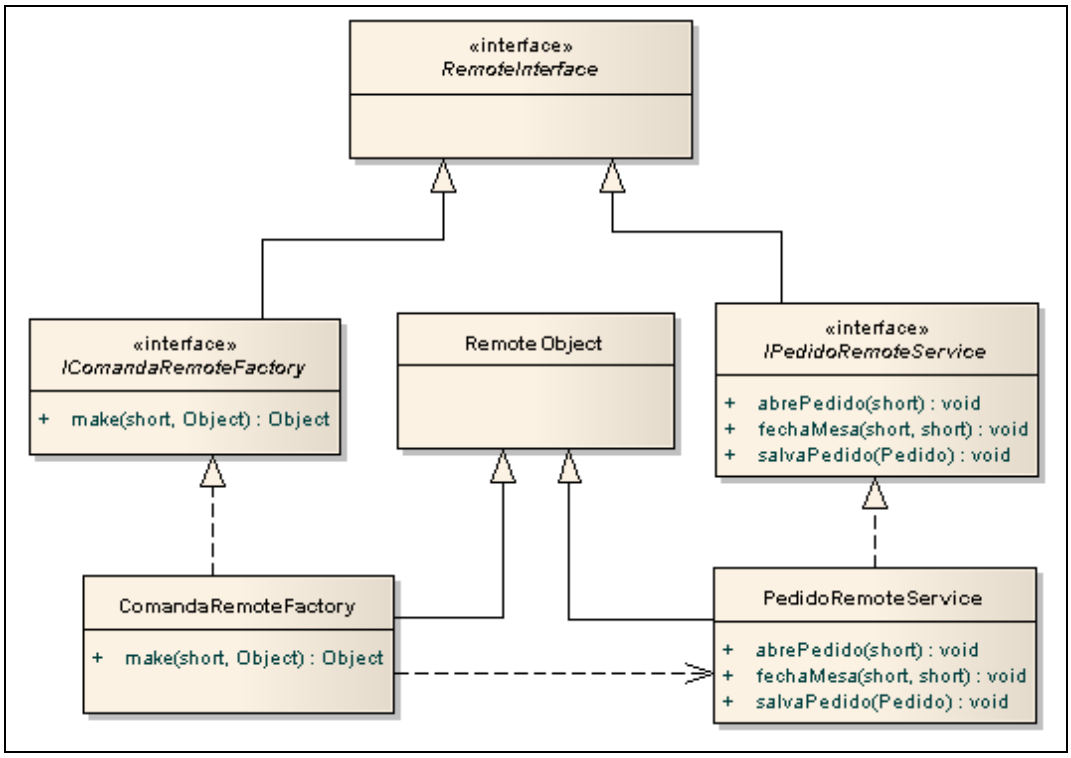


Figura 21 - ComandaRemoteFactory e seus relacionamentos

Como toda classe de objeto remoto, a classe `ComandaRemoteFactory` realiza a interface `RemoteInterface` e estende a classe `RemoteObject`. Além disso, realiza a sua própria interface remota definida por IComandaRemoteFactory. A classe ComandaRemoteFactory possui o método make, utilizado para instanciar objeto de outras classe de objeto remoto.

A Figura 21 apresentada ainda a classe `PedidoRemoteService` cujas instâncias serão criadas por ComandaRemoteFactory. Por se tratar de classe de objeto remoto, `PedidoRemoteService` também realiza uma interface remoto (`IPedidoRemoteService`) e estende `RemoteObject`.

3.4.4 Implementação de uma aplicação distribuída

Para que uma aplicação seja iniciada no servidor de aplicações, deve-se configurar o arquivo `HMI.xml`, informando qual classe é responsável por iniciá-la como mostrado no Quadro 23.

```
<hmi app-class="br.com.ctx.comanda.ComandaAPP"
       timeout="600"
       xdr="br.com.ctx.io.SerializavelXDR"
/>
```

Quadro 23 - Arquivo HMI.xml

O atributo `app-class` nomeia a classe por onde a aplicação é iniciada. A classe de inicialização deve obrigatoriamente estender a classe abstrata `Application` e implementar o método `start`, como mostrado no Quadro 24. O atributo `timeout` indica o tempo, em segundos, que um contexto permanece ativo se não ocorrer atividade na rede. O atributo `xdr` indica qual classe é responsável pela representação externa dos dados durante a transmissão dos dados.

```
package br.com.ctx.comanda;

import br.com.ctx.hmi.*;
import br.com.ctx.hmi.server.*;

public class ComandaAPP extends Application {

      public void start() throws HMIException {
            bind("REMOTE_FACTORY",new ComandaRemoteFactory());
      }
}
```

Quadro 24 - Implementação da classe `ComandaAPP`

No método `start`, são registrados (`bind`) os objetos remotos que serão acessados pelas aplicações clientes. No caso da CE, é registrada somente uma instância de `ComandaRemoteFactory` com o nome `REMOTE_FACTORY`. Este nome é utilizado pela aplicação cliente para localizar (`lookup`) o objeto remoto.

3.4.5 Implementação de uma classe de objeto remoto

As classes de objeto remoto implementam uma interface remota e estendem a classe abstrata `RemoteObject`. No Quadro 25 é apresentado o código da interface remota de

ComandaRemoteFactory, registrada na inicialização da aplicação e responsável pela criação dos demais objetos remotos da aplicação comanda.

```
public interface IComandaRemoteFactory extends RemoteInterface {

   Object make(short service, Object parametros) throws HMIException,
CTxException;

}
```

Quadro 25 - Interface remota de ComandaRemoteFactory

A interface remota de ComandaRemoteFactory possui apenas um método. Todos os métodos de uma interface remota devem prever o lançamento da exceção HMIException. Esta exceção é lançada quando ocorre alguma falha na rede ou não configuração do *middleware* que impeça o seu funcionamento.

Definida a interface remota de ComandaRemoteFactory, escreve-se sua implementação, como mostrado no Quadro 26

```
public class ComandaRemoteFactory extends RemoteObject implements
IComandaRemoteFactory {
   public Object make(short service,Object parametro) throws HMIException,
CTxException {
       switch (service) {
              case ConstantesComanda.LOGIN:
                 return new ComandaSession(parametro);
              case ConstantesComanda.PEDIDO:
                 return new PedidoRemoteService((ComandaSession)parametro);
              case ConstantesComanda.REPOSITORIO_OBSERVACAO:
                 return new ObservacaoRemoteRepository((ComandaSession)parametro);
              case ConstantesComanda.REPOSITORIO_PRODUTO:
                 return new ProdutoRemoteRepository((ComandaSession)parametro);
              default:
                    throw new CTxException ("Classe desconhecida");
       }
   }

}
```

Quadro 26 - Implementação da classe ComandaRemoteFactory

O objetivo de ComandaRemoteFactory é instanciar diferentes tipos de objetos, de acordo com o parâmetro service do método make. Como todo objeto remoto desenvolvido com o *middleware,* a classe ComandaRemoteFactory implementa sua interface remota e estende RemoteObject.

Uma das classes de objeto remoto instanciadas por ComandaRemoteFactory é a PedidoRemoteService, responsável de realizar as tarefas da emissão do pedido que precisam ser executadas no servidor. O código do Quadro 27 apresenta a implementação desta classe.

```
public class PedidoRemoteService extends RemoteObject implements
IPedidoRemoteService   {
       private ComandaSession sessao;

       public PedidoRemoteService(IComandaSession sessao) {
              this.sessao =  (ComandaSession) sessao;
       }

       private IRetaguarda getRetaguarda() throws IOException, CTxException {
              return sessao.getRetaguarda();
       }


       public void abrePedido(short mesa) throws CTxException, IOException {
              getRetaguarda().abrePedido(mesa);
       }

       public void fechaMesa(short mesa, short nroPessoas) throws IOException,
CTxException {
              getRetaguarda().fechaMesa(mesa, nroPessoas);

       }
       public void salvaPedido(Pedido pedido) throws IOException, CTxException {
              getRetaguarda().finaliza(pedido);

       }
}
```

Quadro 27 - Implementação da classe `PedidoRemoteService`

A exemplo da classe `ComandaRemoteFactory`, `PedidoRemoteService` também implementa uma interface remota e estende `RemoteObejct`.

3.4.6 Geração de `Stubs` e `Skeletons`

Um objeto remoto não pode ser acessado remotamente de forma direta. Para que isso seja possível é necessário gerar uma referência remota (*`Stub`*) e um *`Skeleton`*. Os códigos destas classes são gerados através da ferramenta `HmiCompiler`, cuja sintaxe é apresentada no Quadro 28.

```
Java -Dout=dirSaida br.com.ctx.hmi.compiler.HmiCompiler classe1 [classe2, classe3..]

Exemplo:
Java -Dout=C:/temp br.com.ctx.hmi.compiler.HmiCompiler  ComandaRemoteService
PedidoRemoteService
```

Quadro 28 - Sintaxe da ferramenta `HmiCompiler`

Para cada classe passada como parâmetro para `HmiCompiler` são criados os respectivos `Stub` e `Skeleton`. A classe *stub* tem o mesmo nome da classe do objeto remoto com o sufixo `_HmiStub` o `skeleton` tem o mesmo nome, com o sufixo `_HmiSkeleton`. O Quadro 29

apresenta o código fonte do *stub* da classe `ComandaRemotefactory` gerado pela ferramenta `HmiCompiler`.

```
public final class ComandaRemoteFactory_HmiStub extends Stub implements
IComandaRemoteFactory {

   // Stubs podem ser serializados via "writeObject"
   private static final long serialVersionUID = 1L;

   // Construtor para serialização
   public ComandaRemoteFactory_HmiStub() {}

   // Construtor parametrizado
   public ComandaRemoteFactory_HmiStub(br.com.ctx.hmi.RemoteRef remoteRef) {
       super(remoteRef);
   }

   // ID -1301179154 "make"
   public Object make(short arg0, Object arg1) throws HMIException, CTxException {
       try{
              return this.call( -1301179154, new Object[]{new Short(arg0), arg1});
       }
       catch (Throwable t){
              if (t instanceof HMIException){
                     throw (HMIException)t;
              }
              if (t instanceof CTxException){
                     throw (CTxException)t;
              }
              throw new HMIException(t);
       }
   }
}
```

Quadro 29 - Stub da classe ComandaRemoteFactory

Como observado no Quadro 29, o *stub* implementa a mesma interface remota do objeto remoto. Contudo, o código de seus métodos não executam a função implementada naquele objeto e sim, submetem uma requisição ao módulo de comunicação do *middleware*.

Em tempo de execução, o *stub* sempre é instanciado no servidor e enviado para o cliente quando qualquer requisição tem um objeto remoto como retorno. A execução do stub ocorre sempre no cliente. O *skeleton* é o complemento do *stub* no servidor. O Quadro 30 apresenta o *skeleton* da classe `ComandaRemoteFactory` gerado pela ferramenta `HmiCompiler`.

```
public final class ComandaRemoteFactory_HmiSkel extends Skeleton {

	// ID -1301179154 "make"
	private Object make(ComandaRemoteFactory ro,short arg0, Object arg1) throws
		HMIException, CTxException {
		return ro.make(arg0, arg1);
	}

	// implementação do método "dispatch"
	public Object dispatch(RemoteObject ro, int methodId, Object arg) throws
		HMIException {
		Object result = null;
		Object[] args = null;
		try{
			switch (methodId) {
				case -1301179154 : // ID -1301179154 "make"
					args = (java.lang.Object[]) arg;
					result = this.make((ComandaRemoteFactory)ro,
((Short)                                                    args[0]).shortValue(),
args[1]);
					break;
			}
		}
		catch(Throwable t){
			t.printStackTrace();
			if (t instanceof br.com.ctx.hmi.HMIException  ) {
				throw (br.com.ctx.hmi.HMIException )t;
			}
			throw new br.com.ctx.hmi.HMIException (t);
		}
		return result;
	}
}
```

Quadro 30 - Skeleton de ComandaRemoteFactory

O *middleware* não conhece a interface remota dos objetos remotos. Por isso, invoca o método `dispatch` do *skeleton*, passando o objeto remoto e os parâmetros do método que deve ser chamado. O *skeleton* conhece a interface do objeto remoto a partir do qual ele foi gerado e tem como chamar o método responsável por tratar a requisição do *middleware.*

3.4.7 Implementação de objetos serializáveis

Alguns serviços remotos da aplicação CE possuem métodos que têm objetos Java comuns nos parâmetros ou retorno. Considera-se objeto comum todo que não é objeto remoto. É o caso do método `salvaPedido` do objeto remoto `PedidoRemoteService`. Para este método é passado uma instância da classe `Pedido`. Nas plataformas JSE, quando um parâmetro ou retorno não é objeto remoto, uma cópia do objeto é serializada, submetida à rede e recuperada no servidor. Porém a plataforma JME, na qual a camada cliente da aplicação CE é executada, não suporta serialização. JME não tem suporte para transmissão de objetos.

Somente tipos primitivos e Strings são suportados.

Para resolver esta limitação com o mínimo de impacto na semântica nativa da linguagem, os objetos que precisam ser transmitidos entre os nós da aplicação devem implementar a interface `Serializavel`. O Quadro 31 trás um segmento do código fonte da classe pedido, apresentando a implementação desta interface.

```
public class Pedido extends Domain implements Serializavel {

                        . . .

      private static final long serialVersionUID = 1L;
      private int mesa;
      private List itens;

                        . . .

      public void restaura(DataInputStream in) throws IOException {
            mesa = in.readInt();
            itens = (List) Serializador.getInstance().restaura(in);

      }

      public void serializa(DataOutputStream out) throws IOException {
            out.writeInt(mesa);
            out.write(Serializador.getInstance().serializa(itens));

      }
```

Quadro 31 - Implementação de `Serializavel` na classe Pedido

Os métodos restaura e serializa do Quadro 31 mostram a classe Pedido provendo seu próprio recurso de serialização.

3.4.8 Distribuição da aplicação CE

Por se tratar de uma aplicação distribuída, a aplicação CE tem seu componentes instalados em diversos nós de processamento. A Figura 22 apresenta o diagrama de implantação da aplicação.

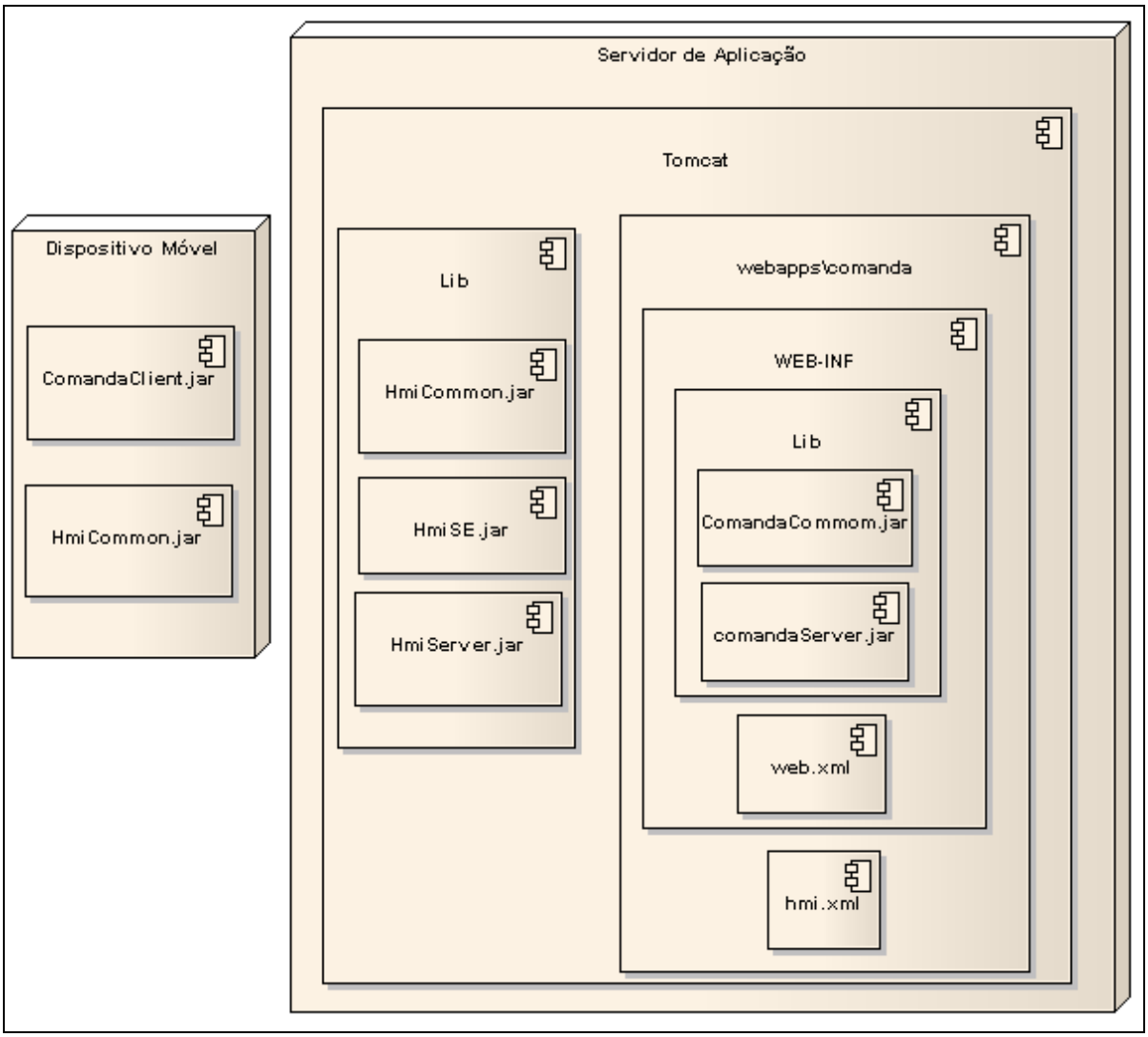


Figura 22 - Diagrama de implantação da aplicação CE

No servidor, a distribuição de aplicações distribuídas com o *middleware* HMI é semelhante ao padrão do *container* que a suporta, no exemplo o *Tomcat*, a exceção de que não é necessário realizar configurações no `web.xml`. Para que a aplicação possa ser acessada é necessário que o *Tomcat* esteja rodando e sua ativação ocorre quando recebe a primeira requisição.

No dispositivo móvel CE é implantada e ativada é como qualquer aplicação Java.

3.4.9 Localização e chamada de objeto remoto

O Quadro 32 mostra o código cliente responsável pela localização do objeto remoto `ComandaRemoteFactory`, registrado na inicialização da aplicação do servidor.

```
public IComandaRemoteFactory getRemoteFactory() throws CTxException, HMIException {
       if (remoteFactory == null) {
              final String propriedade = "HOST";
              final String url = System.getProperty(propriedade);
              if (url == null) {
                     throw new CTxException("Propriedade de sistema não informada: "
                                          + url);
              }
              Context c = Context.getContext(url,80,"comanda");
              remoteFactory = (IComandaRemoteFactory) c.lookup("REMOTE_FACTORY");
       }
       return remoteFactory;
}
```

Quadro 32 - Localização de um objeto remoto

Primeiramente, a aplicação cliente obtém um contexto da aplicação no servidor e em seguida, solicita a localização do objeto remoto, informando o nome com o qual o mesmo foi registrado no servidor.

Após obter uma referência remota para a `factory`, a aplicação cliente chama seus métodos para criar instâncias remotas dos serviços desejados como mostra o Quadro 33.

```
private IPedidoRemoteService getPedidoRemoteService() throws CTxException, IOException
{
  if (pedidoRemoteService == null) {
     ComandaRemoteFactory factory = getRemoteFactory();
     pedidoRemoteService = (IPedidoRemoteService) factory.make(PEDIDO,getSession());
  }
  return pedidoRemoteService;
}
```

Quadro 33 - Chamada de método remoto

A aplicação cliente chama o método `make` do objeto remoto `factory` para criar uma instância remoto de `PedidoRemoteService` e obter uma referência remota para ela, atribuindo-a à variável de instância `pedidoRemoteService`. Observa-se que a sintaxe da invocação do método remoto é exatamente a mesma para chamadas locais. Analisando o código do Quadro 33 de forma isolada é impossível identificar o que é chamada local e o que é remoto. Assim, o requisito de transparência do *middleware* é atingido.

3.5 RESULTADOS E DISCUSSÃO

Este trabalho partiu da dificuldade prática de se construir sistemas distribuídos na web, dentro do paradigma da POO.

Os *middlewares* orientados a objetos, como Java RMI, utilizam um *range* de portas de comunicação. Como na web estas portas são bloqueadas pelos *firewalls* e a utilização destes *middlewares* fica inviabilizada. Além disso, não há implementação do Java RMI para JME,

universo com grande demanda para sistemas distribuídos. Por outro lado, alternativas como os *Web Services*, são adaptados à web, mas não são alinhados com a POO. Chamadas a *Web Services* são orientadas a procedimento e não trata-se de objetos distribuídos. *Web Services* tem ainda necessidade de reconfiguração no *Web Services Descriptor Language* (WSDL) no servidor sempre que um novo serviço é disponibilizado.

O *middleware* objeto deste trabalho une as facilidades do *Java RMI* e dos *Web Services* e é compatível com JME, rodando assim em pequenos dispositivos móveis. Novos objetos remotos podem ser adicionados à aplicação sem nenhuma reconfiguração no *middleware.* Para um objeto remoto ser instanciado, basta que sua classes esteja no *classpath* da máquina virtual, como qualquer objeto Java.

Até onde a pesquisa para fundamentação teórica alcançou, este trabalho é o primeiro *middleware* para objetos distribuídos na *web* e também o primeiro suportado por JME.

O Quadro 34 apresenta uma comparação entre o Java RMI, *Web Servises*, CORBA e este trabalho (HMI).

	Orientado a Objetos	**Transparente Firewalls**	**Compatível JME**	**Linguagens**	**Callback**
Java RMI	SIM	NÃO	NÃO	Java	SIM
Web Service	NÃO	SIM	Com adaptações	Diversas	NÃO
CORBA	SIM	NÃO	NÃO	Diversas	SIM
HMI	SIM	SIM	SIM	Java	NÃO

Quadro 34 - Comparação entre Java RMI, Web Services e HMI

Web Services exige adaptações para ser utilizado em JME por ser baseado em SOAP e depender de um *parser* XML, recursos não disponíveis no JME nativo.

Por questões de tempo, HMI foi implementado apenas em Java. Mas sua especificação é compatível com qualquer linguagem orientada a objeto.

Ainda por questões de tempo, não foi implementado o recurso de *callback*. *Callback* permite que o servidor chame um objeto remoto instanciado no cliente. Este recurso é útil para a implementação de sistemas que utilizam observadores ou notificações.

Este trabalho está sendo aplicado com resultados satisfatórios em sistemas comerciais. A aplicação Comanda Eletrônica é um produto comercializado da Cechinel Tecnologia. O *middleware* também é o suporte para distribuição de objetos no software TMSLog, um sistema para gestão de transporte e logística, em desenvolvimento pela Gertran Tecnologia.

4 CONCLUSÕES

É consenso que o paradigma da análise e programação orientados a objetos trouxe grande contribuição ao desenvolvimento de sistemas. A partir deste paradigma, desenvolveram-se pesquisas e práticas de grande impacto como os padrões de projetos, a UML, o processo unificado, dentre outros. As linguagens que abraçaram a POO, como Java e .Net, dominam as principais iniciativas no desenvolvimento de novos softwares.

A linguagem Java apresenta, desde sua primeira versão, recursos para que se possa estender a POO aos sistemas distribuídos que é o Java RMI. Porém, o modelo adotado pelo Java, baseado em transmissões em formatos binários e em portas de comunicações específicas, encontrou uma forte barreira na sua aplicação na *web* que são os *firewalls*. Por outro lado, Java não possui suporte para objetos distribuídos em dispositivos móveis, justamente onde este recurso é mais necessário.

Para resolver os problemas de incompatibilidade das tecnologias de objetos distribuídos com a web, surgiram os *Web services*. *Web services* resolvem as questões de bloqueios dos *firewalls* e tornaram-se um padrão nos sistemas distribuídos na *web*. Mas não são orientados a objetos. Isto cria uma mistura de paradigmas. O arquiteto de software tem que considerar no seu projeto, componentes que não são objetos e portanto, não têm estados, limitando-se a um conjunto de procedimentos. Isto, além de criar confusão de conceitos durante a análise, causa dificuldades de implementação e manutenção, pois o código se alterna entre trechos orientados a objetos e procedurais.

Da vivência prática das dificuldades expostas acima, surgiu a idéia de se implementar um *middleware* que unisse o paradigma da POO à transparência de passagem pelos *firewalls* apresentada pelos *Web services.* Buscou-se ainda uma solução “leve” para ser suportada por dispositivos móveis.

O *middleware* resultado deste trabalho tem apresentado bons resultados frente aos objetivos a que se propôs. Como demonstrado na operacionalidade da implementação, há uma abstração das questões de rede envolvidas na distribuição dos objetos e como todas as requisições ocorrem em HTTP, não há riscos de bloqueios por *firewalls.* Atingiu-se também o objetivo de se ter um *middleware* compatível com dispositivos móveis.

Em relação aos *Web services*, este trabalho apresenta ainda a facilidade de não exigir reconfigurações quando uma nova classe de objeto remoto (serviço) é adicionada. Basta que a mesma esteja no *classpath* da máquina virtual.

4.1 EXTENSÕES

Sugere-se as seguintes extensões para continuidade do trabalho:

a) adicionar à ferramenta `HmiCompiler` recurso para gerar *stubs* em outras linguagens, além do Java. Isto permitiria que objetos remotos fossem referenciados por clientes desenvolvidos em outras linguagens. Acredita-se ser particularmente interessante, a geração de *stubs* para PHP;
b) aplicar o recurso de código dinâmico da linguagem Java para geração de *Stubs* e *Skeletons* em tempo de execução;
c) implementar *callback*.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott W. **Análise e projeto orientados a objeto:** seu guia para desenvolver sistemas robustos com tecnologias de objetos. Tradução Oswaldo Zanelli. Rio de Janeiro: Infobook, 1998.

APACHE TOMCAT. [S.l.], 2009. Disponível em: <http://tomcat.apache.org/>. Acesso em: 14 fev. 2009.

BARR, Michael**. Toward a Smaller Java**. [S.l.], 2002. Disponível em: <http://www.embedded.com/columns/technicalinsights/9900680?_requestid=40716>. Acesso em: 10 out. 2008.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML**: guia do usuário. Tradução Fábio Freitas. Rio de Janeiro: Campos, 2002.

CONNOLY, Dan**. Hypertext Transfer Protocol**. [S.l.], 1999. Disponível em: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>. Acesso em: 10 out. 2008.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Sistemas distribuídos**: conceitos e projeto. 4. ed. Tradução João Tortello. Porto Alegre: Bookman, 2007.

EHNEBUSKE, David et al. **Simple Object Access Protocol (SOAP) 1.1**. [S.l.], 2000. Disponível em: <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#RFC2774>. Acesso em: 1 set. 2007.

ENTERPRISE ARCHITECT. Austrália, 2008. Disponível em: <http://www.sparxsystems.com.au/about.html>. Acesso em: 25 set. 2008.

FIELDS, Duane K.; KOLB, Mark A. **Java server pages**: um guia prático para projetar e construir serviços dinâmicos na Web. Tradução Rejane Freitas. Rio de Janeiro: Ciência Moderna, 2000.

GAMMA, Erich et al. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Tradução Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000.

HERITY, Dominic**. Multicores need a distributed object model framework**. [S.l.], 2006. Disponível em: <http://www.embedded.com/columns/embeddedsoapbox/175803345?_requestid=63382>. Acesso em: 23 ago. 2008.

JAVA REMOTE METHOD INVOCATION SPECIFICATION. Califórnia, 2004. Disponível em: <http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf>. Acesso em: 20 ago. 2008.

JAVA REMOTE METHOD INVOCATION. [S.l.], 2008. Disponível em: <http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp#1>. Acesso em: 16 ago. 2008.

JAVA TECHNOLOGY OVERVIEW. [S.l.], 2008. Disponível em: <http://java.sun.com/overview.html>. Acesso em: 23 ago. 2008.

JENDROCK, Eric et al. **The java enterprise editon tutorial**. Califórnia, 2007. Disponível em: <http://java.sun.com/javaee/5/docs/tutorial/doc>. Acesso em: 28 ago. 2008.

JUNIOR, Décio. **HTTP**: guia de consulta rápida. São Paulo: Novatec, 2001.

LEE, Valentino; SCHNEIDER, Heather; SCHELL, Robbie. **Aplicações móveis**: arquitetura, projeto e desenvolvimento. Tradução Amauri Bentes e Deborah Rüdiger. São Paulo: Pearson, 2005.

LOUREIRO, Antonio ª F. et al. **Comunicação sem fio e computação móvel**: Tecnologias, Desafios e Oportunidades. São Paulo, 2007. Disponível em <http://homepages.dcc.ufmg.br/~loureiro/cm/docs/jai03.pdf>. Acesso em: 12 set. 2008.

LOUREIRO, Emerson C. L. F. **Middleware extensível para disponibilização de serviços em ambientes pervasivos**. 2006. 115 f. Dissertação (Mestrado em Informática) - Curso de Pós-Graduação em Informática, Universidade Federal de Campina Grande, Campina Grande.

MAHMOUD, Qusay. **J2ME APIs**: which APIs come from the J2SE platform? [S.l.]: Sun Microsystems, 2001. Disponível em: <http://developers.sun.com/mobility/midp/articles/api/>. Acesso em: 23 ago. 2008.

MUHAMMAD, Hisham H. **Exploração de reflexão computacional através de um modelo de objetos sem classes.** São Leopoldo, 2003. Disponível em <http://www.sbc.org.br/reic/edicoes/2003e3/cientificos/Exploracao_de_reflexao_computacional_atraves_de_um_modelo_de_objetos_sem_classes.pdf>. Acesso em: 15 set. 2008.

ORB BASICS. [S.l.], 2007. Disponível em: <http://www.omg.org/gettingstarted/orb_basics.htm>. Acesso em: 23 ago. 2008.

PAVAN, Willingthon. **Técnicas de salvamento e recuperação de estados de objetos**. Porto Alegre, 1998. Disponível em: <http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana98/pavan.html>. Acesso em: 12 set. 2008.

PEREIRA, Fernando M. Q. **Chamada remota de métodos na plataforma J2ME/CLDC**. 2003. 129 f. Dissertação (Mestrado em Ciências da Computação) – Curso de Pós-Graduação em Ciências da Computação, Universidade Federal de Minas Gerais, Belo Horizonte.

RICHARDSON, Chris. **Pojos em ação**: como desenvolver aplicações corporativas com frameworks leves. Tradução Marcelo Trannin Machado. Rio de Janeiro: Ciência Moderna, 2007.

ROCHA, C. Utilizando web services em aplicações móveis. **Web mobile**, Rio de Janeiro, n. 16, p. 16-25, 2007.

SCOTT, Kevin. **Habilidades de TI que você precisa ter**. [S.l.], 2007. Disponível em: <http://computerworld.uol.com.br/gestao/2007/07/24/idgnoticia.2007-07-23.2421306793/>. Acesso em: 11 ago. 2007.

TANENBAUM, Andrew S. **Sistemas operacionais modernos**. Tradução Ronaldo A. L. Gonçalves. São Paulo: Pearson, 2007.

TANENBAUM, Andrew S.; STEEN, Maartin V. **Sistemas distribuídos**: princípios e paradigmas. Tradução Arlete Simille Marques. São Paulo: Pearson, 2008.

THE JAVA SERVLET API WHITE PAPER. [S.l.], 1998. Disponível em: <http://java.sun.com/products/servlet/whitepaper.html>. Acesso em: 23 ago. 2008.

WEB SERVICES ACTIVITY. [S.l.], 2002. Disponível em: <http://www.w3.org/2002/ws/>. Acesso em: 2 set. 2007.