Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

MCF: Um Framework de Conectividade para Aplicações Móveis em Java ME

Carlos Eduardo dos S. P. Silva

Trabalho de Graduação

Recife
3 de Abril de 2007

Universidade Federal de Pernambuco
Centro de Informática

Carlos Eduardo dos S. P. Silva

MCF: Um Framework de Conectividade para Aplicações Móveis em Java ME

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Prof. Geber Ramalho*

Recife
3 de Abril de 2007

Agradecimentos

A Deus e a minha família, principalmente os meus pais, que proporcionaram a minha formação.

A Tereza, minhha namorada, que me acompanha em todos os momentos, sejam eles bons ou ruins. E que, mesmo não sendo muito compreensiva, gosta muito de mim.

Ao meu orientador Geber, pelo direcionamento e por ter confiado em mim para a realização deste trabalho.

A Rafael Borges, por ter sido meu co-orientador, e que me apoiou bastante, sempre disposto a me ajudar no desenvolvimento do trabalho.

Aos membros do extinto grupo mobili, Bruno Pereira e Geraldo Fernandes, por me ajudarem a desenvolver meu gosto pela programação de jogos e aplicações móveis.

A Meantime Mobile Creations e ao C.E.S.A.R, empresas onde dei o primeiro passo em minha carreira profissional. Ao pessoal do projeto 3MOG, pela amizade e troca de experiências, que serviram de base para a realização deste trabalho.

Finalmente, à turma de computação 2006-2, que passou junto comigo por toda esta jornada acadêmica.

Resumo

O aumento de poder computacional dos dispositivos móveis os tornam cada vez mais próximos dos computadores pessoais e promoveram o surgimento de plataformas para desenvolvimento de aplicações, como Brew e Java ME, que entre outras vantagens oferecem suporte à aplicações conectadas. Entretanto, apenas a presença de conectividade sem o auxílio de uma biblioteca de programação em alto-nível traz muito mais complexidade para as aplicações, aumentando o tempo de desenvolvimento, os custos e a propensão a erros. Tal fato realça a necessidade de um componente reutilizável e extensível que torne ágil o desenvolvimento desta classe de aplicações. Diante do contexto apresentado, o objetivo deste trabalho é propor a construção de um framework de conectividade para aplicações móveis de propósitos gerais. O framework em questão será responsável pelos aspectos de baixo nível relativos à conectividade, direcionando os esforços do desenvolvedor para o processamento da lógica inerente à aplicação.

Palavras-chave: Java ME, Aplicações conectadas, framework de conectividade

Abstract

The increase in computational power of the mobile devices brings them close to personal computers and also promoted the creation of application development platforms like Brew and Java ME, which among other advantages, support connected applications. However, the presence of conectivity itself without aid of a high-level programming library brings more complexity for the applications, increase the development time, costs and the probability of errors. This fact suggest the need of a reusable and extensible component that makes the developement of this class of applications more agile. In the context presented here, the objective of this work is to propose the construction of a connectivity framework for general purpose mobile applications. This framework will be responsible for the low-level aspects related to connectivity, directing the developer's efforts to the applications' logic.

Keywords: Java ME, Connected applications, Connectivity framework

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

Os dispositivos móveis estão cada vez mais presentes no cotidiano. A rápida evolução do poder de processamento destes dispositivos os aproximam cada vez mais dos computadores pessoais. Além disso, a melhoria da capacidade de conectividade destes dispositivos, proporcionada tanto pela maior quantidade de dados trafegados nas redes de telefonia quanto pelo surgimento de tecnologias de comunicação sem fio como Bluetooth[1], Wi-fi[2] e WiMax[3], os tornam uma plataforma peculiar, onde a informação pode ser acessada a qualquer momento e em qualquer lugar.

Serviços conectados que antes eram possíveis apenas através de PCs como email, *e-banking* e *e-commerce* passaram a ser acessadas também por estes dispositivos, e proporcionaram o surgimento de novos modelos de negócio. As aplicações de *streaming* de mídia e os jogos móveis também têm encontrado nos dispositivos móveis uma nova plataforma de conectividade. O aumento da largura de banda e diminuição da latência das redes torna cada vez mais próxima a consolidação dos jogos móveis multiusuário, que também se beneficiam da criação de redes *ad-hoc*, onde a latência e a largura de banda se aproximam das experimentadas pelas redes fixas.

Esta crescente evolução possibilitou a execução de aplicações cada vez mais complexas e despertou interesse na indústria de desenvolvimento de *software* levando ao surgimento das plataformas de desenvolvimento Brew[4] e Java ME[5].

BREW - Binary Runtime Enviroment for Wireless é uma plataforma para dispositivos móveis criada pela Qualcomm[6]. Além de uma plataforma de desenvolvimento e um ambiente de execução de aplicações, BREW oferece também serviços de distribuição, testes e certificação de aplicações. Embora tenha sido criada para telefones celulares CDMA, esta plataforma é independente da rede de telefonia utilizada, e pode ser também utilizada em telefones GSM. BREW é particularmente popular nos EUA e Japão, enquanto no Brasil está disponível apenas em celulares da VIVO[7]. Dentre as desvantagens de BREW está o modelo de negócios fechado, onde o desenvolvedor precisa comprar licenças para executar as aplicações nos dispositivos e estas precisam passar por testes e certificações da Qualcomm. Este modelo torna a competição ainda mais difícil para os desenvolvedores independentes e empresas menores, com baixo orçamento e pouco tempo. Além disso, a maioria das ferramentas de desenvolvimento de apli-

[1] http://www.bluetooth.com/bluetooth/
[2] http://www.wi-fi.org/index.php
[3] http://www.wimaxforum.org/home/
[4] http://brew.qualcomm.com/brew/en/
[5] http://java.sun.com/javame/index.jsp
[6] http://www.qualcomm.com/
[7] http://www.vivo.com.br/portal/home.php

cações em BREW (que normalmente utilizam C ou C++) são proprietárias e os dispositivos habilitados têm muitas incompatibilidades.

Por sua vez, Java ME é uma plataforma de desenvolvimento voltada para variados dispositivos com restrições de memória e processamento, tais como celulares, PDAs e set-top boxes. Criada pela Sun[8], Java ME é um subconjunto da versão voltada para aplicações em computadores pessoais, o Java SE, mantendo grande parte de seus conceitos e de sua sintaxe. Para abarcar melhor a imensa quantidade de dispositivos nesta categoria, a especificação utiliza os conceitos de configurações e perfis. As configurações definem o subconjunto Java básico disponível e os recursos da máquina virtual a ser utilizada, enaquanto os perfis disponibilizam bibliotecas específicas para uma determinada classe de dispositivos. Esta abordagem facilita a adoção da plataforma tanto pelo mercado quanto pelos desenvolvedores.

Neste trabalho, consideramos Java ME mais promissor por ser hoje considerada um padrão no mercado, uma vez que está habilitada em milhões de dispositivos móveis com as mais diferentes características. Os jogos móveis e as aplicações corporativas encontraram nesta tecnologia um nicho promissor a ser explorado. Devido à inerente conectividade dos dispositivos móveis, principalmente os telefones celulares, e da evolução das redes de telefonia já citada, há uma tendência no desenvolvimento de jogos conectados e jogos móveis multiusuário além da utilização dos dispositivos móveis como clientes de aplicações que antes eram acessadas por *browsers* e aplicações *desktop*.

Entretanto, Java ME ainda enfrenta muitas dificuldades em relação ao desenvolvimento de aplicações conectadas. Particularmente, o *GCF - Generic Connection Framework*, que provê uma forma unificada de acesso às bibliotecas que implementam os protocolos de conectividade suportados pelos dispositivos, oferece uma *API* de baixo nível, orientada a *bytes*; cabe ao desenvolvedor realizar o controle do fluxo de envio e recepção e a interpretação dos dados obtidos. É importante destacar que este controle de fluxo deve ser realizado através de *threads* separadas, desobstruindo a linha principal de execução. Há então também a necessidade de controle de concorrência, caso contrário a aplicação poderá se comportar diferente do esperado.

Estes fatores deixam claro que a presença de conectividade sem o auxílio de uma biblioteca de programação em alto-nível traz complexidade desnecessária para as aplicações, aumentando o tempo e o custo de desenvolvimento, além da quantidade de defeitos [Bou05]. É evidente, portanto, a necessidade de um componente reutilizável e extensível que torne ágil o desenvolvimento desta classe de aplicações.

1.1 Objetivos

O objetivo deste trabalho é propor a construção do *MCF - Mobile Connectivity Framework*, um *framework*[9] de conectividade para aplicações móveis. O *MCF* é um *framework* extensível orientado a eventos, que provê uma camada de abstração para o *GCF* e é responsável pelos aspectos de baixo nível relativos à conectividade e controle de concorrência. O *MCF* simplifica

[8] http://www.sun.com/

[9] *Frameworks* são uma implementação básica e reutilizável de uma arquitetura de software composta de pontos de extensão. Pontos de extensão são determinados locais do código onde o desenvolvedor insere lógica específica de sua aplicação.

a comunicação entre a aplicação cliente e o servidor, direcionando os esforços do desenvolvedor para o processamento da lógica inerente à aplicação. O desenvolvimento do *MCF* aplica os princípios de orientação a objetos com o objetivo de fornecer uma implementação com alto grau de abstração, extensível e de fácil manutenção.

O *framework* construído faz uso intensivo de criação de objetos e quando utilizados em dispositivos com pouca memória disponível, o coletor de lixo é executado de forma mais freqüente, o que consome mais tempo de processamento. Ao analisarmos sua implementação, percebemos que vários objetos têm curto tempo de vida ou funcionam como ponto de sincronização entre *threads*. Através de otimizações de código, chega-se a uma abordagem que minimiza a criação e a sincronização desnecessária de objetos no *MCF*, embora esta abordagem seja mais baixo-nível, exigindo mais trabalho por parte do desenvolvedor.

Diante destas duas abordagens de desenvolvimento apresentadas, para ter melhor proveito das vantagens de cada uma, este trabalho sugere um mapeamento entre ambas. Para amenizar este problema, deve-se minimizar a criação de objetos e o descarte instantâneo de objetos que não estão mais sendo utilizados. Devido à relevância deste problema, será proposta uma modelagem altenativa do *framework*, que exige menos memória e processamento, embora evite utilizar grande parte dos conceitos de orientação a objetos.

É importante ressaltar que este trabalho é apenas um passo inicial no sentido do desenvolvimento de aplicações móveis conectadas, onde podem ser explorados, por exemplo, aspectos específicos dos jogos móveis multiusuário e de aplicações em tempo real.

1.2 Visão Geral

No capítulo 2 serão apresentados os desafios relacionados ao desenvolvimento de aplicações móveis conectadas, o problema a ser tratado neste trabalho. Ainda neste capítulo serão mostrados os requisitos para o desenvolvimento de componentes que venham a facilitar o desenvolvimento desta classe de aplicações. Em seguida, no capítulo 3 serão estudadas as características e utilizações dos *frameworks* no desenvolvimento de aplicações. A primeira contribuição do trabalho aparece no capítulo 4, onde é especificado e modelado o *MCF*. O capítulo 5 mostra um estudo de caso da utilização do *MCF* no desenvolvimento do jogo OthelloME. A segunda contribuição dá-se no capítulo 6, onde será mostrada uma nova abordagem de implementação do *MCF*, otimizada para menor consumo de memória e processamento, visando atender os requisitos dos dispositivos móveis mais limitados. Os trabalhos futuros e as considerações finais serão apresentadas no capítulo 7.

CAPÍTULO 2

O Problema

O desenvolvimento de aplicações móveis conectadas é uma tarefa complexa, onde o programador, além de se preocupar com a lógica da aplicação (fluxo de controle, estados, estruturas de dados), deve também se preocupar com o mecanismo de envio e recebimento de dados através da rede, tarefa que exige conhecimentos específicos, que vão além das regras de negócio da aplicação. Além disto, aplicações móveis possuem requisitos extremamente restritos em relação a poder de processamento e de capacidade de memória, onde toda a sua lógica deve ser executada em apenas algumas dezenas ou centenas de *kilobytes*.

As redes de conectividade por sua vez ainda apresentam elevada latência na transmissão dos dados [Nok04b], muitas vezes através de conexões intermitentes e com uma elevada perda de pacotes [Wu04], o que dificulta o desenvolvimento de aplicações que utilizem transmissão de dados em tempo real. Soma-se a isto o fato de que tanto os dispositivos móveis quanto as redes de telefonia são bastante heterogêneos: diferentes protocolos de rede são suportados por diferentes redes e dispositivos móveis, o que dificulta bastante a portabilidade das aplicações.

A necessidade de contornar a maioria destes problemas tende a elevar os custos no desenvolvimento de aplicações móveis conectadas, onde muitas vezes as soluções utilizadas em uma aplicação, normalmente custosas, não são reaproveitas em outras, aumentando o *time-to-market*. Estes fatores deixam clara a necessidade de se utilizar um componente de rede reutilizável que isole os aspectos de conectividade de baixo nível.

Neste capítulo apresentaremos as motivações para o desenvolvimento de aplicações móveis conectadas. Em seguida analisaremos o suporte à conectividade oferecido por Java ME, identificando seus pontos fortes e limitações. Por fim, levantaremos os requisitos que devem ser atendidos por uma solução que trate dos aspectos de conectividade desta classe de aplicações.

2.1 Motivação

No inicio de sua existência, os serviços de telefonia móvel tinham como foco principal a comunicação em voz através de sistemas analógicos. Após a migração destes sistemas para aqueles que utilizam transmissão digial, os serviços móveis passaram também a transmitir dados além de voz. A partir de um desenvolvimento contínuo, as redes móveis evoluíram, partindo das tecnologias GSM, GPRS, CDMA até chegar na 3ª geração (3G)[Wu04], onde cada vez mais a latência dimunui, enquanto a largura de banda aumenta. Paralelamente, as tecnologias de redes sem fio (Bluetooth, Wi-fi, WiMax) são suportadas por um número crescente de dispositivos, possibilitando diferentes modelos de conectividade, como a criação de redes *ad-hoc*.

Devido à inerente conectividade dos dispositivos móveis, as aplicações conectadas que an-

tes eram possíveis apenas através de PCs, passaram a ser acessadas também por estes dispositivos. Serviços como email, *m-commerce*, *m-banking* e *m-Learning*, já estão disponíveis para os usuários de sistemas móveis, criando novos modelos de negócio [WA+03, RT+06]. Aplicações multimídia, incluindo vídeo sob demanda, *streamming* de áudio e até voz sobre IP também têm encontrado nos dispositivos móveis uma nova plataforma de conectividade.

Na área dos jogos móveis, há uma tendência para a popularização de um novo nicho, o dos jogos móveis multiusuário. Incentivados tanto pela capacidade de transmissão de dados através da Internet quanto pela possibilidade da criação de redes ad-hoc (Bluetooth, Wi-fi), estes jogos permitem que os jogadores utilizem seus dispositivos móveis para disputar partidas com outros usuários, submeter e consultar o *rank* de um jogo através da Internet ou até interagir com jogos multiusuário de PCs. Há ainda iniciativas na indústria do desenvolvimento dos jogos móveis massivamente multiusuário, onde centenas ou até milhares de jogadores interagem através de seus telefones celulares em um mundo de jogo compartilhado [Pal03, AM+06b, AM+06a].

Tais fatores deixam claro que as aplicações conectadas são um caminho natural na evolução dos dispositivos e redes móveis. Devido à presença em larga escala destes dispositos, que são inerentemente conectados, a indústria tem direcionado sua atenção para este mercado, que ainda pode ser muito promissor. Para o meio acadêmico este ainda é um campo de estudo incipiente, com inúmeros desafios a serem estudados.

2.2 Conectividade em Java ME

A conectividade de Java ME é oferecida através do *GCF - Generic Connecion Framework* [Sun03]. *GCF* é uma hierarquia de interfaces e classes utilizada para criar conexões e processar entrada e saída de dados. O *GCF* provê uma abordagem genérica à conectividade, pois fornece uma biblioteca de classes fundamentais a todos os tipos de conexões, desde as baseadas no envio e recebimento de dados em blocos (pacotes), até as que transmitem os dados em um fluxo contínuo (*streams*).

A generalização do *GCF* é fornecida através do uso de:

- Uma hierarquia extensível de interfaces;
- Uma fábrica de conexões;
- URLs para indicar os tipos de conexões a serem criadas.

Em geral os requisitos de conectividade e armazenamento persistente dos dispositivos móveis variam de um para o outro. Enquanto um celular pode possuir conectividade apenas através de HTTP e *sockets TCP*, um outro celular ou PDA pode dar suporte a datagramas e *Bluetooth*. Esta heterogeneidade de suporte a protocolos de rede dificulta o trabalho do programador, principalmente se o mesmo precisa se preocupar com características de baixo nível dos protocolos.

A idéia básica do *GCF* é ter uma classe, *Connector*, que possa criar qualquer tipo de conexão através de uma *URL*. Este procedimento é realizado através do método estático *open*, que é mostrado abaixo.

```
Connector.open("esquema:endereço;parâmetros");
```

Por exemplo, para realizar uma conexão HTTP ao servidor java.sun.com e solicitar o recurso /javame, devemos utilizar a seguinte *URL*:

```
Connector.open("http://java.sun.com/javame");
```

De forma bastante similar, para acessar um arquivo chamado *entrada.txt* utilizamos o seguinte:

```
Connector.open("file://entrada.txt");
```

A flexibilidade do GCF está na interpretação dos protocolos em tempo de execução: *Connector* procura uma classe apropriada que implemente o protocolo solicitado; Se a classe for encontrada, será retornado um objeto que implementa a interface *Connection*. Desta forma, um dispositivo só precisa possuir as classes que implementam os protocolos por ele suportados. A classe *Connector* e a interface *Connection* são definidos na CLDC [Sun03]. A hierarquia de conexões é mostrada na figura 2.1.

No GCF, se dois protocolos diferentes possuem funcionalidades em comum, tais fucionalidades são acessadas de forma unificada. Vale ressaltar que nenhum protocolo de rede é definido pela CLDC, e sim através dos diversos perfis, de acordo com as necessidades.

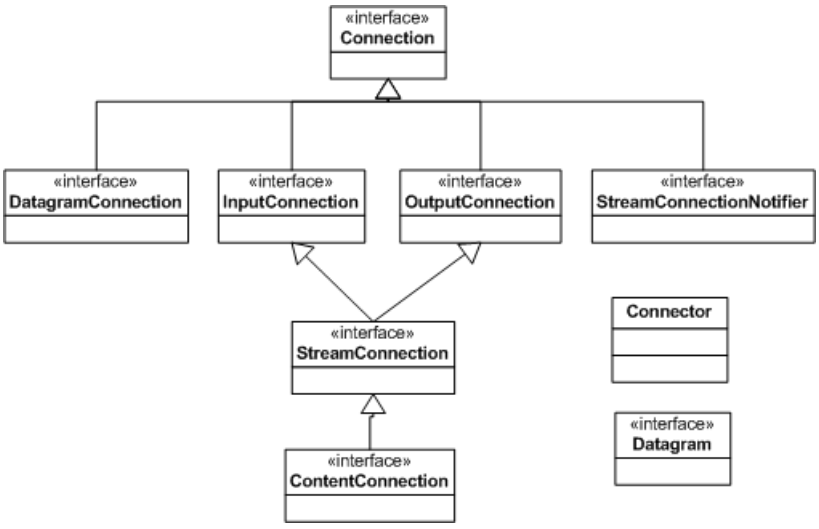

Figura 2.1 Hierarquia de interfaces do GCF

2.2.1 Limitações

Conforme visto na seção anterior, Java ME provê suporte extensível a conectividade através do *GCF*. Porém, o *GCF* utiliza uma abordagem de baixo nível, onde mensagens são enviadas

e recebidas através da rede na forma de fluxos de bytes. Cabe ao desenvolvedor da aplicação gerar ou interpretar estes dados de maneira que sejam úteis à aplicação, normalmente através de mapeamento em objetos Java.

Outra limitação deve-se ao fato que o envio e recebimento dos dados é feito através de operações bloqueantes, o que impossibilita a aplicação de realizar outras tarefas enquanto está conectando ou enviando e recebendo dados da rede. Também não é fornecido um modelo de conectividade onde as operações de rede executam em processos paralelos, deixando a aplicação livre ações, tais como mostrar informações visuais sobre o estado atual da transmissão dos dados ou permitir que o usuário interrompa a transmissão antes da mesma ter acabado. Fica a cargo do programador a criação e sincronização destes processos paralelos.

Em resumo, o *GCF* possibilita a utilização em Java ME dos diversos protocolos de rede suportados por um determinado dispositivo; porém, cabe ao programador o gerenciamento da conexão e o envio e recebimento dos dados através da manipulação de fluxos de bytes e o controle de concorrência. Esta abordagem é complicada e propensa a erros.

2.3 Requisitos para o desenvolvimento de aplicações móveis conectadas

Devido à importância dos problemas levantados anteriormente, tanto os inerentes à classe de aplicações estudada, quanto os que surgem através da utilização do *GCF*, identificamos as principais necessidades no desenvolvimento de uma solução que trate dos aspectos de conectividade das aplicações móveis, de maneira ágil e sem muito comprometimento nos custos do desenvolvimento.

Reuso

O reuso é uma forma de utilizar partes de um *software* ou de conhecimento do domínio para a construção de novos *softwares*. O reuso pode ser feito através de código-fonte (trechos de código, *templates*, procedimentos ou bibliotecas), modelagem (padrões de projeto) ou por uma combinação de ambos. Na indústria, o reuso tem trazido uma série de benefícios no desenvolvimento de projetos como maior produtividade, maior confiabilidade e melhores estimativas de custos [G+95]. A utilização de uma solução de conectividade para aplicações móveis que seja reaproveitável, portanto, traz todos estes benefícios.

Programação em alto nível

Paradigmas de programação em alto nível, a exemplo da orientação a objetos, fornecem mecanismos para que aspectos de baixo nível das aplicações sejam encapsulados (ou abstraídos) na forma de classes, *APIs*, etc. Desta forma pode-se utilizar entidades abstratas, sem a necessidade de suas respectivas implmentações. A orientação a objetos também facilita o reuso, inclusive através da construção de *frameworks*. Portanto, a utilização de orientação a objetos pode nos oferecer:

- Maior facilidade no desenvolvimento e extensão das funcionalidades;

- Menor necessidade de conhecimento específico do domínio de redes e conectividade móvel

Baixo *footprint*

Devido às restrições já mencionadas dos dispositivos móveis, principalmente em relação a memória e processamento, o desenvolvedor de aplicações móveis deve ter constante cuidado com a quantidade de memória alocada e com o desempenho da aplicação. Grande parte das aplicações móveis são instaladas através de *downloads* na Internet e a largura de banda disponível na maioria das redes móveis ainda são baixas, portanto deve-se ter atenção em relação ao tamanho do pacote de instalação das aplicações (contendo os códigos binários, imagens, sons) que muitas vezes não podem ultrapassar algumas dezenas de *kilobytes*.

A presença de conectividade em aplicações móveis leva naturalmente a aumentos no número de classes, de objetos alocados e também ciclos de processamento na aplicação. Estes fatores devem ser pensados, principalmente quando as aplicações devem executar em dispositivos com severas restrições.

CAPÍTULO 3

Frameworks

Segundo [Joh97], um *framework* orientado a objetos é uma arquitetura reusável de todo ou parte de um sistema, que é representado por um conjunto de classes abstratas e pela forma como as suas instâncias interagem. *Frameworks* provêem uma solução reusável, para uma família de problemas semelhantes. Seu conjunto de classes deve ser flexível e extensível para permitir a construção de várias aplicações com pouco esforço, especificando apenas as particularidades de cada aplicação. Através da utilização de *frameworks* o custo do desenvolvimento de aplicações pode ser reduzido drásticamente.

Frameworks diferem de outras técnicas de reuso, como a utilização de componentes de *software* e a de padrões de projetos. A componentização é baseada na reutilização de código, enquanto os padrões de projeto se baseiam na reutilização de *design*. Já os *frameworks* constituem uma solução que, além de englobar várias idéias de *design* também possuem solução na forma de código. De fato, um *framework* muitas vezes é composto de diversos padrões de projetos e componentes.

Muitas vezes os *frameworks* são confundidos com bibliotecas de classes orientadas a objetos. Nas bibliotecas de classes, cada classe independe das outras e a colaboração entre estas é determinada pela própria aplicação, por outro lado nos *frameworks* as colaborações entre as classes que os compõem já são pré-determinadas. Outra diferença é que o *framework* é quem invoca o código específico de uma aplicação, este mecanismo é chamado de *Princípio de Hollywood* [1] [Hau97]. Já nas bibliotecas de classe isto não acontece. A tabela 3.1 destaca as principais diferenças entre estas duas técnicas de reuso.

Bibliotecas	**Frameworks**
classes instanciadas pelo desenvolvedor	customização através de subclasses ou composição
o desenvolvedor chama métodos	chama métodos definidos pelo desenvolvedor
não possui fluxo de controle pré-definido	controla o fluxo da execução
não possui interação pré-definida	define a interação entre os objetos

Tabela 3.1 Diferenças entre frameworks bibliotecas de classes

[1] Também conhecido como "*Don't call us, we'll call you.*", uma referência às respostas clichés dadas aos atores amadores em testes na indústria de cinema. Em programação está relacionado ao modelo de invocações implícitas de métodos.

Hot spots e Frozen spots

Hot spots são os pontos de extensão (classes abstratas, métodos que devem ser implementados) do *framework*. Para um desenvolvimento bem sucedido, deve-se identificar estes pontos de acordo com o domínio das aplicações. *Frameworks* não são aplicações completas; para que isto aconteça, deve-se instanciá-lo através da implementação de cada *hot spot*. Uma vez que estes são implementados, o *framework* executará o código destes pontos de extensão através de chamadas implícitas (lembre-se do princípio de Hollywood). Nestas chamadas, o código implementado pelo desenvolvedor da aplicação é executado na ocorrência de determinado evento.

Algumas funcionalidades dos *frameworks* são imutáveis, por fazerem parte de alguma decisão de projeto utilizada na arquitetura dos mesmos. Estes pontos imutáveis constituem o núcleo de um *framework*, também conhecidos com *frozen spots*. *Frozen spots*, ao contrário dos *hot spots*, são pedaços de código já implementados no *framework*, que chamam um ou mais *hot spots* implementados pelo desenvolvedor da aplicação. O núcleo sempre será uma constante e estará presente em qualquer instância do *framework*.

Frameworks de caixa-branca

Um *framework* é do tipo caixa-branca (Em inglês, *white-box*)[RJ97] se seus pontos de extensão se baseiam na utilização de herança e de *overriding* de métodos. O termo *caixa-branca* se refere à visibilidade interna do *framework*, que devido à herança, os elementos internos das classes-pai são normalmente visíveis para as subclasses.

Entre as vantagens dos *frameworks* de caixa-branca estão:

- a herança de classes é definida estaticamente, em tempo de compilação; portanto é uma abordagem clara e direta;
- A implementação a ser reusada pode facilmente ser modificada, embora não se possa mudar em tempo de execução a implementação das classes derivadas;
- Tendem a ser soluções bastante flexíveis pois o desenvolvedor pode modificar seus componentes de forma que os seus projetistas nunca antes imaginaram.

Contudo, estes tipos de *frameworks* apresentam algumas desvantagens, onde a principal delas diz respeito à dificuldade de aprendizado, pois para saber como utilizá-los é necessário conhecer como o *framework* foi construído. Além disto, a implementação de uma subclasse é diretamente ligada à implementação de sua classe-pai. Esta dependência limita a flexibilidade e o reuso. Outro fator adverso, devido ao excesso de generalização, é a grande quantidade de código que deve ser adicionado ao *framework* para se prover alguma funcionalidade útil à aplicação.

Em resumo, um *framework* de caixa-branca nada mais é que um esqueleto de arquitetura onde as classes específicas, definidas pelo desenvolvedor, são adicionadas ao mesmo. Normalmente o processo de desenvolvimento se inicia com uma abordagem de caixa-branca, sendo considerados estes um estágio natural do processo evolutivo de um sistema. À medida em que o sistema evolui, o projetista notará que partes do *framework* deixarão de ser genéricas para se tornarem especializadas.

Frameworks de caixa-preta

Os *frameworks* de caixa-preta se baseiam na composição de objetos. Nesta abordagem, o desenvolvedor especifica as funcionalidades desejadas para a aplicação através da combinação e composição de objetos. A composição de objetos é um mecanismo de extensão mais flexível do que a utilização de herança entre classes, principalmente devido ao fato de que estas composições podem ser modificadas dinâmicamente, enquanto herança é um conceito estático. A composição de objetos utiliza o relacionamento *"Has a"*, no qual um objeto utiliza outro objeto através de uma referência ao mesmo.

A composição de objetos requer que os objetos a serem compostos apresentem interfaces bem definidas para que o desenvolvedor não precise entender os elementos internos de cada objeto e simplesmente utilize as suas funcionalidades.

Dentre as vantagens destes tipos de *framework* estão:

- Os detalhes de implementação dos componentes não são visíveis pelos desenvolvedores. A composição de objetos tende a manter cada objeto encapsulado e especializado em uma determinada funcionalidade, desta forma torna-se mais fácil assimilar os conceitos do *framework*;
- Requer menos programação, pois os componentes apenas necessitam ser combinados, não há necessidade de se derivar classes e implementar métodos;
- Permitem que mudanças no comportamento possam ser feitas em tempo de execução, através da substituição de um componente por outro que possua a mesma interface.

O principal problema enfrentado por esta abordagem é que estes *frameworks* são menos flexíveis, pois o número de combinações possíveis entre os componentes é determinado pela arquitetura do *framework*. Não há outra forma de se definir algum comportamento diferente do que foi planejado inicialmente no projeto do mesmo.

Frameworks de caixa-preta são uma evolução natural dos de caixa-branca [RJ97], pois, à medida que o projeto de um *framework* se torna melhor compreendido, é mais fácil fornecer componentes com mais funcionalidades prontas. Todavia, nem sempre esta transição ocorre, pois alguns *frameworks* de caixa-branca fornecem todas as funcionalidades a que se propõem, e são considerados um produto finalizado.

3.1 Utilização

Um típico *framework* é um esqueleto arquitetural extensível que provê funcionalidades básicas em um domínio específico. Fica a cargo do desenvolvedor da preencher as funcionalidades específicas da aplicação, preenchendo os seus *hot spots*. Nos *frameworks* do tipo caixa-branca este processo é feito através de herança e *overriding* de métodos enquanto nos do tipo caixa-preta a extensão ocorre através das diferentes combinações entre objetos fornecidos. *Frameworks* de caixa-branca tendem a ser difíceis de utilizar pois requerem que os desenvolvedores escrevam uma quantidade significativa de código para produzir o comportamento esperado. Já

os de caixa-preta muitas vezes podem ser limitados, por possuírem um conjunto de classes que não permitem extensão. É frequente a utilização de abordagens híbridas, possuindo herança e *overriding* além de alguma funcionalidade pré-definida.

A utilização de um *framework* pronto traz benefícios facilmente perceptíveis em termos de redução dos custos no desenvolvimento e do *time-to-market*. Através da utilização de *frameworks* o reuso é levado ao extremo, onde são reaproveitados não só o código mas também toda a análise do problema, projeto e testes destas soluções. Os desenvolvedores passam a direcionar seus esforços à lógica específica da aplicação ao invés de precisarem reinventar soluções em aplicações do mesmo domínio. Pelo fato dos *frameworks* fatorarem os aspectos comuns a diversos tipos de aplicações, fica mais fácil de se fazer manutenções nestas aplicações, se ambas utiliarem o mesmo *framework*, além de proporcionar maior consistência e compatibilidade entre estas aplicações.

Devido ao uso extensivo em diversas aplicações, os *frameworks* são soluções seguras, pois na maioria das vezes já foram testados e validados em aplicações anteriores e possuem um código estável, menos propenso a defeitos.

Um dos problemas relacionados à utilização de *frameworks* está na sua curva de aprendizagem. Estas soluções normalmente ditam padrões de programação, com quais as aplicações precisam se adaptar. Além disto, muitas vezes é necessário ter conhecimento dos relacionamentos entre as suas classes, para que se possa utilizá-los. Torna-se evidente a importância de uma boa documentação para o *framework*, que por sua vez, também não é um processo simples, pois *frameworks* são mais abstratos do que a maioria das aplicações. Diversas abordagens são pesquisadas para documentá-los da forma mais clara e fácil [Joh96, But]. Estas dificuldades fazem com que a maioria dos especialistas acreditem que a maneira mais fácil de se obter conhecimento sobre um *framework* é através de sua utilização.

3.2 Desenvolvimento

O desenvolvimento de *frameworks* é uma tarefa complexa e custosa que exige do programador, entre outras qualidades, o conhecimento do domínio e de *design*. Por este motivo, *frameworks* devem ser desenvolvidos apenas quando forem utilizados em diversas aplicações de um mesmo domínio, conpensando os custos de seu desenvolvimento.

Os *frameworks* devem ser soluções simples e fáceis de aprender, porém devem também prover as funcionalidades mais importantes para o domínio a que eles se propõem. Em [G+95] é apontada a utilização de padrões de projeto como chave do sucesso na construção de *frameworks* com um elevado nível de reuso de *design* e de código. Os padrões de projeto também se mostram bastante úteis na documentação dos *frameworks*. Por serem soluções já conhecidas e bem aceitas, tornam-se uma linguagem de entendimento comum para explicar as idéias utilizadas no *framework*.

[RJ97] sugere a utilização de uma abordagem iterativa e evolutiva no desenvolvimento de *frameworks*, onde um desenvolvedor, por melhor que seja, não pode simplesmente pensar e surgir com uma solução abstrata para um *framework* que resolva um determinado domínio de problemas, a menos que ele já tenha vasta experiência no desenvolvimento de aplicações deste domínio.

Um bom método de iniciar o desenvolvimento é o padrão chamado de "Three Examples", que propõe o desenvolvimento de três aplicações nas quais supõe-se que o *framework* irá ajudar a desenvolvê-las. Desenvolve-se a primeira aplicação, em seguida desenvolve-se uma segunda aplicação que seja um pouco diferente da primeira, por fim, desenvolve-se uma terceira aplicação que seja um pouco diferente das outras duas. Dado que estas aplicações estão em um mesmo domínio, abstrações em comum irão surgir facilmente.

Este é apenas um início de um processo evolutivo, que é mostrado na figura 3.1, onde o *framework* começa como uma solução de caixa-branca e evolui até se tornar uma solução de caixa-preta. Entre estes dois estágios, existem outros intermediários como objetos plugáveis, *hot spots* e bibliotecas de componentes. Por fim o *framework* talvez chegue em ponto onde pode ser construído através de ferramentas visuais, através das quais seus componentes podem ser conectados para produzir o resultado desejado e em seguida o código do *framework* é gerado. Porém, nada impede de que um *framework* de caixa-branca já seja considerada uma solução madura e finalizada.

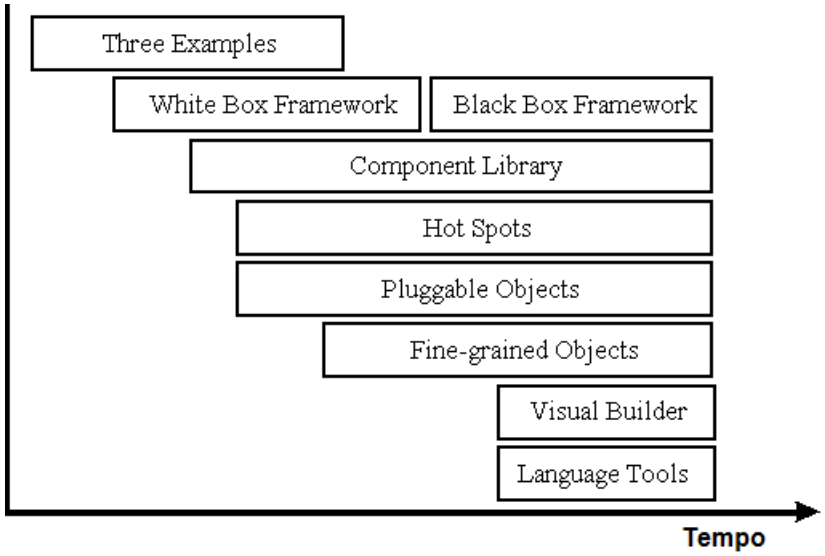

Figura 3.1 Relacionamento entre as fases evolutivas de um framework

CAPÍTULO 4

MCF - Mobile Connectivity Framework

Neste capítulo nós propomos a construção do *MCF*, um *framework* de conectividade para aplicações móveis, que foi projetado para tornar o desenvolvimento destas aplicações menos oneroso, pois o *MCF* se encarrega dos aspectos de rede de baixo nível e do controle de sincronização em relação ao envio e recebimento de dados através da rede. Com isto, o desenvolvedor pode direcionar seus esforços à implementação da lógica da aplicação, e apenas estender o *framework* de forma que ele se adeque ao protocolo de comunicação utilizado entre a aplicação e o servidor.

Neste capítulo, primeiramente definiremos os requisitos a serem levados em consideração no desenvolvimento do *MCF*, tais requisitos foram baseados em experiências anteriores de desenvolvimento de aplicações móveis conectadas. Em seguida explanaremos sobre o método utilizado na modelagem e implementação do *MCF*. Finalmente, apresentaremos a modelagem proposta para o *framework* em questão, assim como o modelo de sincronização entre seus compontentes que nós utilizaremos.

4.1 Requisitos

A implementação do *MCF* contempla a classe das aplicações móveis conectadas que são tolerantes a atraso (aplicações em tempo real não serão contempladas neste trabalho) porém intolerantes à perda de pacotes. A aplicação que utiliza o *MCF* necessita da garantia de que qualquer mensagem trocada com o servidor não se perca na rede, a não ser que a conexão com o servidor falhe completamente. Isto contrasta com aplicações tolerantes a perdas (aplicações de *streaming* de áudio em tempo real), onde pacotes podem ser descartados por conterem informações que já não são mais úteis no momento em que chegaram.

Comunicação assíncrona

Em um modelo síncrono de comunicação, a aplicação requisita dados, espera pelo resultado e só então os processa. Já no modelo assíncrono, a aplicação requisita os dados e posteriormente é notificda quando os dados estiverem prontos para serem processados. Neste meio tempo a aplicação pode fazer qualquer outra requisição ou processamento, o que torna a aplicação mais interativa.

No contexto do *MCF*, as mensagens enviadas ao servidor devem ser não bloqueantes, isto é, no instante em que uma mensagem é enviada, a aplicação deve estar livre para enviar uma nova mensagem ou executar qualquer outra operação. Isto possibilita chamadas do tipo *fire*

and forget onde a aplicação simplesmente envia dados ao servidor e não necessariamente está esperando alguma resposta do mesmo.

Orientação a eventos

Um evento pode ser entendido como um pedaço de informação relevante à aplicação, que pode indicar uma mudança de estado, uma resposta a alguma solicitação ou uma notificação. Na programação orientada a eventos, a aplicação possui um *loop* que espera continuamente pela chegada de algum evento para ser processado. Quando isto acontece, é disparada um *handler* responsável por processá-lo, executando as devidas atualizações na aplicação, possivelmente extraindo informações do evento. Programação orientada a eventos não é nada mais que definir *handlers* para tratar cada tipo de evento relevante à aplicação.

As mensagens trocadas entre o *MCF* e o servidor da aplicação devem ser tratadas como eventos. Quando chega alguma informação do servidor, o *framework* notifica a aplicação através de um evento que a aplicação sabe como processá-lo. Este modelo de programação, além de permitir uma forma de comunicação assíncrona, possibilita o desacoplamento entre o código de rede e a lógica da aplicação e permite uma maior flexibilidade para aplicação, pois para se tratar um novo evento basta apenas criar um novo *handler* e registrá-lo como responsável por tratar um determinado evento. Cada *handler* tende a ter um alto grau de independência em relação aos demais, incentivando uma programação mais limpa que alivia o trabalho do programador, evitando que ele se preocupe com o fluxo no código. Este modelo possibilita uma abordagem que condiz com o princípio de *Hollywood*, mostrado no capítulo 3.

Sockets TCP

A classe de aplicações tratadas pelo *MCF*, por ser intolerante à perda de pacotes, necessita de garantia de entrega das mensagens trocadas com o servidor. Desta forma seria dispendioso o uso do protocolo *UDP*, pois o *framework* precisaria fornecer um mecanismo de identificação de erros e garantia de entrega dos pacotes, funcionalidades que já são fornecidas pelo protocolo TCP.

Através da utilização de TCP como protocolo de transporte, a comunicação com a rede pode ser realizada através de HTTP ou de *sockets TCP*. A primeira abordagem utiliza um modelo de requisição/resposta, onde para se trocar dados entre o cliente e o servidor, o cliente envia uma mensagem de requisição de um recurso e o servidor envia uma mensagem de resposta ao cliente com a solicitação. A versão do HTTP presente na grande maioria dos dispositivos móveis é a 1.0, na qual o tempo de vida da conexão está restrito a cada requisição. Porém, cada vez que uma conexão é estabelecida ou encerrada, é consumida uma grande quantidade de tempo da CPU, de largura de banda e de memória.

Por outro lado, ao utilizar *sockets* TCP diretamente, a conexão é estabelecida apenas uma vez e a aplicação pode enviar e receber dados simultaneamente através da utilização de canais *full-duplex*. Os *sockets* TCP também permitem a utilização do modelo *push-based*, onde o servidor pode enviar dados a qualquer hora para a aplicação, mesmo que ela não os tenha requisitado.

Devido aos fatos apresentados acima, a utilização de *sockets* TCP é a abordagem que melhor

se encaixa na classe de aplicações tratadas neste trabalho.

4.2 O Método Utilizado

A idéia por trás do *MCF* surgiu a partir do projeto MMMOG[1], uma parceria entre a FINEP[2], o C.E.S.A.R[3] e a Meantime Mobile Creations[4] com o propósito de desenvolver uma plataforma servidora para jogos móveis massivamente multiusuário [Pal03, AM+06b, AM+06a].

Durante o projeto, foram desenvolvidos alguns jogos multiusuário e também aplicações clientes para validar a plataforma. Entre estes jogos, destaca-se o *Hangout*, um jogo móvel multiusuario de interação social, onde centenas de jogadores interagem entre si em diversos locais de um mundo compartilhado. Os custos com a implementação das funcionalidades de conectividade de *Hangout* tornaram visível a necessidade de se utilizar um *framework* para acelerar seu desenvolvimento. Ao mesmo tempo, a experiência adquirida com este desenvolvimento sem reuso direcionou a identificação dos aspectos que podem ser abstraídos e automatizados.

4.2.1 Generalização

A modelagem e implementação das primeiras aplicações, incluindo o *Hangout*, que serviram como ponto inicial para a abstração dos conceitos de conectividade, foram baseadas em [Nok04a]. Esta abordagem é mostrada no diagrama da figura 4.1, onde as classes responsáveis pela conectividade estão destacadas. A classe *TCPHandler* é responsável pela conexão TCP, além de ser usada para transmitir e receber mensagens da rede. Quando *TCPHandler* recebe alguma mensagem, invoca o método *handleMessage* da classe que implementa a interface *TCPHandlerListener*, que seria a classe responsável por atualizar o estado da aplicação. O método *handleMessage* já apresenta características similares ao princípio de *Hollywood*, pois é invocado implicitamente pelo componente responsável pela rede.

O envio de mensagens através da rede era feito através do método *queueMessageForSending*, que recebia uma cadeia de bytes. Ficava a cargo do programador preencher esta cadeia de bytes de acordo com o protocolo da aplicação. Este não era um processo prático e não fugia muito da abordagem de baixo nível oferecida pelo *GCF* de Java ME.

A partir daí, resolvemos separar o tratamento da rede em três partes. A primeira seria responsável pelo envio de dados através da rede, a segunda pelo recebimento e processamento destes dados, enquanto a terceira, por tratar dos aspectos relativos à conexão propriamente dita (conectar, desconectar, criar os fluxos de entrada e de saída). O próximo passo foi melhorar o mecanismo de envio de dados, de forma que o desenvolvedor pudesse trabalhar com um nível maior de abstração, facilitando o desenvolvimento e proporcionando maior extensibilidade. O processo de recebimento de mensagens já dava os primeiros passos em relação à extensibilidade e o reuso, porém esta não era uma solução modularizável, que em aplicações mais complexas não apresentaria muitas facilidades. Desta forma generalizamos o modelo de recebimento de

[1] http://sourceforge.net/projects/mmmog/
[2] http://www.finep.gov.br/
[3] http://www.cesar.org.br/
[4] http://www.meantime.com.br/pt/

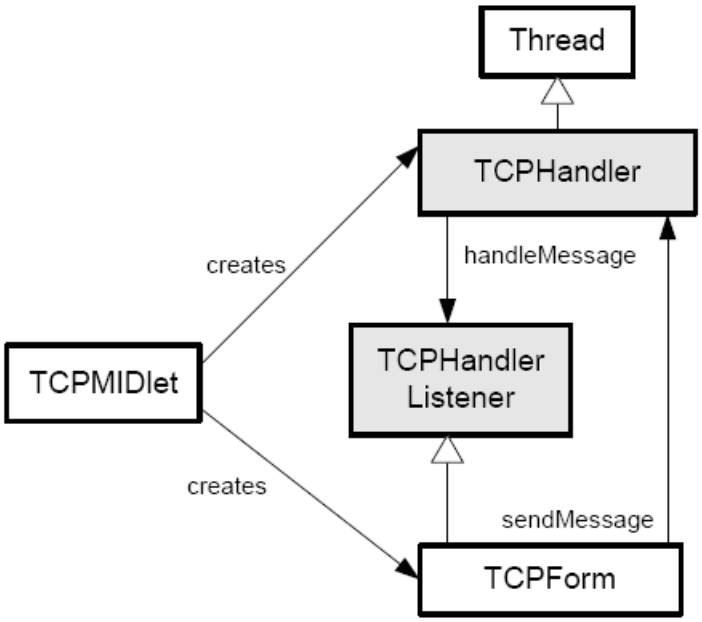

Figura 4.1 Diagrama de classes de uma aplicação com *TCPHandler*

mensagens criando abstrações semelhantes às do envio. Posteriormente ainda dividimos o mecanismo de recebimento e processamento de mensagens em duas partes e conseguimos com isto maior generalização e modularidade.

4.2.2 Proposta

Conforme discutido anteriormente, o *framework* foi desenvolvido através de um processo de generalização das funcionaliades de conectividade de aplicações de um domínio comum. Após discutirmos sua modelagem e implementação, instanciaremos o *framework* para ser utilizado em um estudo de caso (capítulo 5), e com isto poderemos analisar o valor de tais generalizações na prática.

4.3 Modelagem e implementação

Nesta seção explicaremos os detalhes da modelagem e implementação do *MCF*. Dividimos a modelagem em três partes: a primeira trata do modelo proposto para representar as mensagens transmitidas pela rede; a segunda sobre o modo como estas mensagens são enviadas ao servidor e como o *MCF* transforma as mensagens vindas do servidor em eventos da aplicação; por fim, na terceira parte mostramos o modelo de sincronização entre os componentes do *framework*. O *MCF* é desenvolvido como um *framework* de caixa-branca, onde as funcionalidades específicas da aplicação são adicionadas pelo desenvolvedor através de herança e implementação de

métodos.

Assumimos nesta implementação que o *MCF* se comunica com o servidor através de um protocolo binário, onde as mensagens que trafegam na rede possuem formato semelhante ao mostrado na figura 4.2. Aplicações que se comunicam com o servidor através de outros formatos de mensagem (ex. mensagens em XML ou SOAP) não são contempladas neste trabalho. Utilizamos protocolo binário por motivos de economia de consumo de banda e de processamento, recursos escassos nos dispositivos móveis [Nok03].

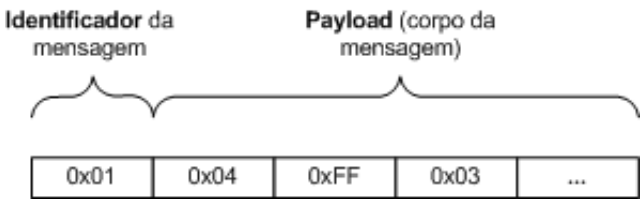

Figura 4.2 Exemplo de um protocolo binário simples

4.3.1 Modelo de mensagens

Na modelagem do *MCF* propomos um modelo onde os fluxos de bytes são representados através de mensagens bem tipadas que são facilmente interpretadas e processadas pela aplicação. A figura 4.3 mostra o diagrama de classes da hierarquia de mensagens.

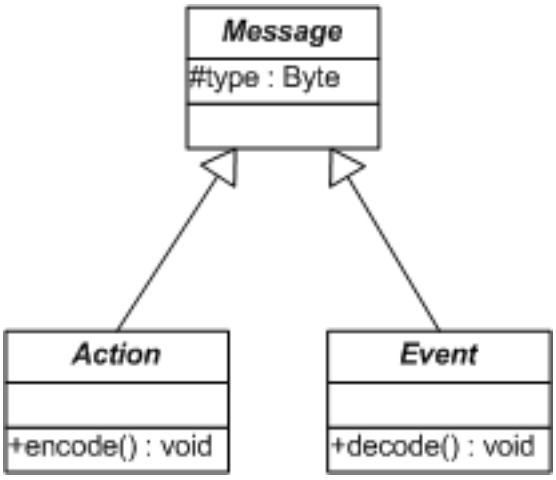

Figura 4.3 Diagrama de classes do modelo de mensagens

Message É uma classe abstrata que representa uma mensagem transmitida na rede, tanto no sentido cliente-servidor quanto no sentido servidor-cliente. Possui um atributo que identifica o tipo de mensagem, de acordo com o protocolo binário a aplicação.

Action Esta classe abstrata, que herda de **Message**, representa uma ação (requisição, envio de dados) no sentido cliente-servidor. Action possui o método *encode*, responsável por converter seus atributos em uma cadeia de bytes a ser transmitido pela rede.

Event Esta classe também é abstrata e estende de **Message**. Representa um evento que chega à aplicação cliente através da rede. Esta classe possui o método *decode* que preenche os atributos da classe, a partir de um fluxo de bytes recebidos da rede, criando um novo evento para a aplicação.

Pontos de extensão Este conjunto de classes abstratas que representam o modelo de mensagens são os primeiros pontos de extensão (*hot spots*) do *MCF*, nos quais o desenvolvedor adiciona funcionalidade específica para as mensagens utilizadas pela aplicação a ser desenvolvida. O desenvolvedor deve estender a classe *Action* para os diversos tipos de ações que a aplicação pode enviar para o servidor e implementar o método *encode()* para cada classe derivada, fazendo com que seus atributos sejam serializados para serem enviados ao servidor. O desenvolvedor também deve criar uma subclasse de *Event* para cada evento que a aplicação pode receber do servidor, e sobrescrever o método *decode* para mapear a transformação do fluxo de bytes recebido da rede em um objeto que representa um evento da aplicação.

4.3.2 Modelo de comunicação

As classes que compõem o modelo de mensagens necessitam de um mecanismo que as transmitam da aplicação cliente ao servidor e vice-versa, e que coordenem o processo de conversão Mensagem-bytes e bytes-Mensagem. Além disto, o *framework* necessita de uma arquitetura que possibilite a comunicação assíncrona baseada em eventos. Para tanto, projetamos a arquitetura do *MCF* em dois processos paralelos, onde um é responsável por controlar o envio de mensagens ao servidor enquanto o outro é responsável por receber e processar eventos provenientes do servidor, e em seguida atualizar o estado da aplicação.

Em [Fer06] é apresentado um modelo de programação orientada a eventos através do padrão de projeto *Extended Handlers*, que é mostrado na figura 4.4. Este modelo é constituído por três componentes: o gerador de eventos, o *dispatcher* e um conjunto de *event handlers*. O gerador de eventos é responsável por gerar os eventos que são utilizados na aplicação. O *dispatcher* recebe cada evento proveniente do gerador de eventos e em seguida analisa-o para determinar o seu tipo e posteriormente enviar cada evento ao *event handler* apropriado. Para que o *dispatcher* possa processar um fluxo contínuo de eventos, a sua lógica deve conter um *loop* de eventos onde ele pode receber um evento, despachá-lo e em seguida processar o evento seguinte. Os *event handlers* contêm a lógica de processamento de seus respectivos eventos.

Em algumas situações pode acontecer do *dispatcher* receber um evento e não encontrar um *handler* apropriado. Neste caso o *dispatcher* pode simplesmente descartar o evento ou lançar uma exceção.

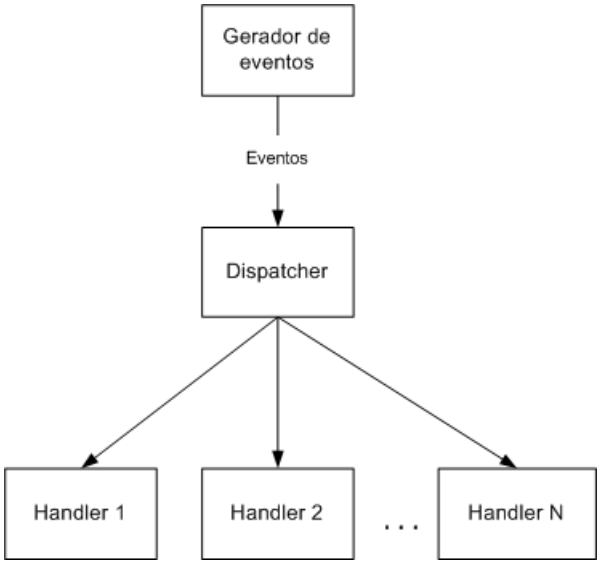

Figura 4.4 O padrão de orientação a eventos Extended Handlers

Envio de ações ao servidor

Aplicações conectadas freqüentemente necessitam solicitar informações ou enviar dados ao servidor. O *GFC* fornece meios para que estes dados sejam transmitidos pela rede através de diversos protocolos, porém, os dados transmitidos precisam estar codificados em cadeias de bytes, uma vez que este é o modelo no qual o *GCF* trata os dados. Existe também a restrição na qual o envio de dados com *GCF* é feito através de operações bloqueantes, ou seja, enquanto os dados estão sendo enviados ao servidor, a *thread* na qual esta operação é executada permanece bloqueada até que o envio se complete. Esta é uma característica não desejável em nossa implementação pois o *framework* tem como requisito a comunicação assíncrona, que no nosso caso se dá através de chamadas não bloqueantes (assíncronas).

Basicamente necessitamos de um mecanismo responsável por enviar os objetos *Action*, que representam uma mensagem destinada ao servidor. Este mecanismo é mostrado na figura 4.5. Para tal finalidade, criamos a classe *EncoderThread*, que herda da classe *thread* e possui a função de *dispatcher*, onde aguarda em um *loop* a chegada de eventos provenientes da aplicação (objetos *Action*) e em seguida chama o *event handler* responsável por processar cada evento específico, transformando os atributos do objeto *Action* em bytes que em seguida serão enviados ao servidor. É importante observar que os *event handlers* estão encapsulados na própria classe *Action*, através do método *encode* e que a classe *EncoderThread* possui o fluxo de saída de dados para o servidor (OutputStream).

Abaixo é mostrado o *loop* da classe *EncoderThread*:

```
while (this.running) {
    // Espera que algum evento seja gerado
    Action action = ...
```

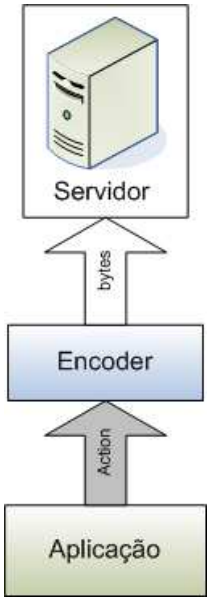

Figura 4.5 O envio de ações ao servidor

```
    // Invoca o event handler do evento
    action.encode(this.output);

    // Envia o fluxo de dados pela rede
    this.output.flush();
}
```

Recebimento e processamento de eventos

O recebimento de dados provenientes do servidor - que pode ser uma resposta a alguma requisição da aplicação ou simplesmente alguma notificação de mudança de estado da aplicação por parte do servidor - também é tratado no *MCF* através de orientação a eventos. Quando algum fluxo de bytes chega na aplicação na aplicação pela rede, o *MCF* se encarrega de interpretar estes dados e transformá-los em eventos que são tratados pela aplicação. A figura 4.6 mostra o esquema de decodificação e geração de eventos no *MCF*.

A classe *DecoderThread*, que tem acesso ao fluxo de entrada de dados, faz o papel de gerador de eventos. É a *thread* responsável por ler as cadeias de bytes provenientes do servidor e convertê-las em objetos **Event**, que serão despachados para serem processados. Através de um *loop*, há uma espera até que haja algum byte disponível no fluxo de entrada. O primeiro byte lido indica o tipo de evento que está sendo recebido. Esta informação é utilizada para criar a subclasse de **Event**, através do padrão de projeto factory [EF+04, G+95], que será responsável por decodificar o restante das informações. Ao invocar o método *decode* da subclasse que representa o evento, seus atributos são preenchidos a partir do consumo das informações presentes no fluxo de entrada (InputStream). Finalmente, o objeto **Event** já criado e preenchido

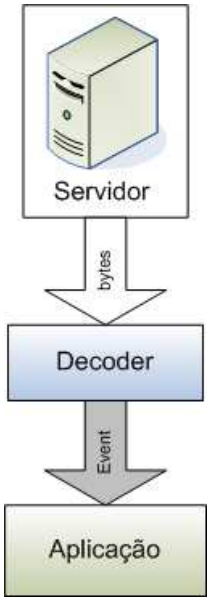

Figura 4.6 O processo de geração e decodificação de eventos no MCF

é enviado para ser processado, que será o passo que explicaremos a seguir.

```
while (true) {
    // Espera até que exista algum dado disponível na rede
    byte eventType = this.input.readByte();

    // Cria a subclasse específica que representa o evento
    Event event =
            (Event) this.messageFactory.create(eventType);

    // Decodifica o resto das informações do evento
    event.decode(this.input);

    // Envia o evento para ser processado
    ...
}
```

Processamento de eventos - O próximo passo após a geração dos eventos é o seu processamento, para que, a partir das informações presentes no evento, o estado da aplicação seja modificado. Este esquema é mostrado na figura 4.7. A *thread* encarregada de controlar este processamento é representada pela classe *ExecutorThread*. Esta *thread*, funciona como um *dispatcher*, onde aguarda em um *loop* o recebimento do eventos e se encarrega de criar um objeto *Executor* capaz de processar cada evento recebido. O *Executor* é um *event handler* que utiliza conceitos similares ao padrão de projeto *Command* [EF+04, G+95], no qual comandos

são encapsulados como objetos para que as classes que desejam executar tais comandos não precisem ter conhecimento da lógica de execução. No nosso caso, A classe *ExecutorThread* simplesmente obtem objetos do tipo *Executor* que sabem como tratar cada tipo de evento.

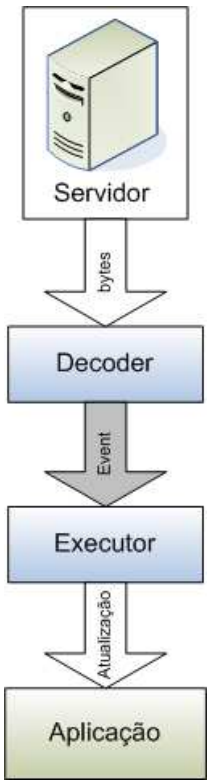

Figura 4.7 O processo de execução dos eventos gerados pelo *decoder*

```
while (this.running) {
    // Espera a chegada de um evento
    Event event = ...

    // Checa o tipo do evento e obtém um Executor adequado
    byte eventType = event.getType();
    Executor executor =
              (Executor) this.executorFactory.create(eventType);

    // Despacha a execução do evento
    executor.execute(event);
}
```

Pontos de extensão O segundo ponto de extensão do *MCF* é a criação das classes responsáveis por tratar os eventos recebidos e fazer as devidas atualizações no estado da aplicação. Este

processo é feito através da implementação da interface *Executor*, criando classes concretas para processar cada um dos objetos Event utilizados na aplicação. *Executor* é uma interface Java que possui apenas o método *execute(Event e)* e funciona como um *event handler*. Na implementação deste método, o desenvolvedor tem acesso às informações do evento a ser processado, e fica a cargo dele obter um meio de acessar o estado da aplicação, para que o mesmo possa ser atualizado. Isto pode ser feito por exemplo através do padrão de projeto *singleton* [EF+04, G+95] ou através de uma camada de indireção. Abaixo é mostrado o código da interface *Executor*, que contém a assinatura do método *execute*.

```
public interface Executor {
    public void execute(Event event);
}
```

4.3.3 Modelo de sincronização

Em programação concorrente, a ordem correta das interações entre processos e a coordenação do acesso aos recursos compartilhados entre estes são fatores de extrema importância para o correto funcionamento da aplicação. Para coordenar o acesso a recursos compartilhados entre si, os processos precisam se comunicar. Esta comunicação poder ser feita através de envio de mensagens ou por memória compartilhada [H+98]. Java possui mecanismos de programação concorrente embutidos na própria sintaxe da linguagem e em sua máquina virtual, onde um processo é representado por uma *thread* e a comunicação entre *threads* é feita através de compartilhamento de memória. Java ME herda grande parte da funcionalidade de programação concorrente (threads, monitores e locks) mas não fornece uma API com classes utilitárias comumente utilizadas em programação concorrente.

Na seção 4.3.2 vimos que, no recebimento e processamento de eventos, as classes *DecoderThread* e *ExecutorThread* se comunicam através do envio de objetos *Event*. Outro exemplo de *threads* comunicantes existentes nesta implementação será mostrado mais adiante. Para controlar a comunicação entre as *threads* que compõem o *MCF* implementamos a classe *BlockingQueue*. Esta é uma das classes que fazem parte do pacote `java.util.concurrent`, introduzido na versão 1.5 do Java SE, mas que não está presente em nenhuma versão do MIDP.

BlockingQueue representa uma fila bloqueante onde a operação de remoção de um elemento bloqueia até que haja algo na fila para ser removido, este processo é mostrado na figura 4.8, enquanto os métodos desta classe são mostrados na tabela 4.1. Assim, sempre que uma *thread* precisa enviar uma mensagem a outra, a primeira enfileira a mensagem em uma *BlockingQueue* na qual a segunda *thread* esteja esperando e esta será imediatamente notificada.

Unindo os componentes

Por fim, é oportuno falar sobre a classe *Client*, que é responsável por inicializar e finalizar cada uma das *threads* do *framework*, além de gerenciar a conexão via *sockets* TCP com o servidor (conexão e desconexão, abertura dos fluxos de entrada e saída).

As operações de conexão e desconexão com a rede são chamadas bloqueantes, e nos dispositivos móveis tendem a demorar alguns segundos para serem executadas. Por este motivo

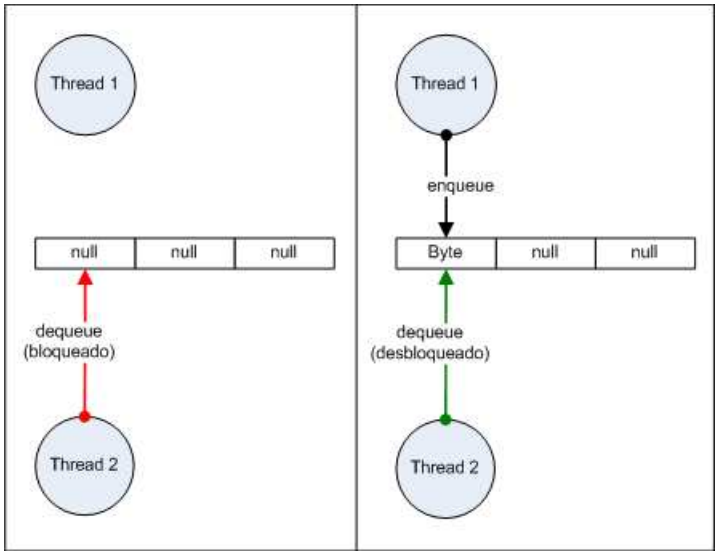

Figura 4.8 Modelo de filas bloqueantes

Método	**Descrição**
enqueue(Object o)	Insere um objeto na fila.
dequeue()	Remove o primeiro elemento da fila. Bloqueia até que haja algum elemento na fila.
dequeue(long timeout)	Remove o primeiro elemento da fila. Bloqueia até que haja algum elemento na fila ou até que o timeout expire.

Tabela 4.1 Métodos da classe *BlockingQueue*

deve-se evitar fazer tais operações na mesma *thread* em que aplicação executa, caso contrário a aplicação ficará travada até que o processo se complete. Por este motivo, o procedimento de abrir e fechar uma conexão, feito pela classe *Client*, executa em uma *thread* separada, desta maneira a aplicação fica livre para dar feedbacks ao usuário, ou inclusive para possibilitar o cancelamento da operação de abrir conexão.

Além de ser responsável pelo gerenciamento, tanto da conexão com o servidor, quanto das *threads* que compõem o *MCF*, esta classe também é a interface de saída entre o *MCF* e aplicação, pois possui o método *send(Action action)*, através do qual a aplicação pode enviar ações ao servidor. O método *send* simplesmente enfileira um objeto **Action** na *BlockingQueue* de saída, que é compartilhada pelo *EncoderThread*. Através de uma operação de *dequeue*, o *EncoderThread* recebe o objeto **Action** e faz os procedimentos de codificação e envio pela rede. Este modelo de envio de mensagens através de filas faz com que o método *send* seja uma operação não bloqueante, o que proporciona a característica de assincronia do *framework*.

A classe *Client* junto com as classes *DecoderThread*, *EncoderThread* e *ExecutorThread* fazem parte do pacote `framework.core`, que agrupa os componentes do *framework* que não são modificados nem estendidos pelo desenvolvedor, os *frozen spots*. O diagrama de classes

deste pacote é mostrado na figura

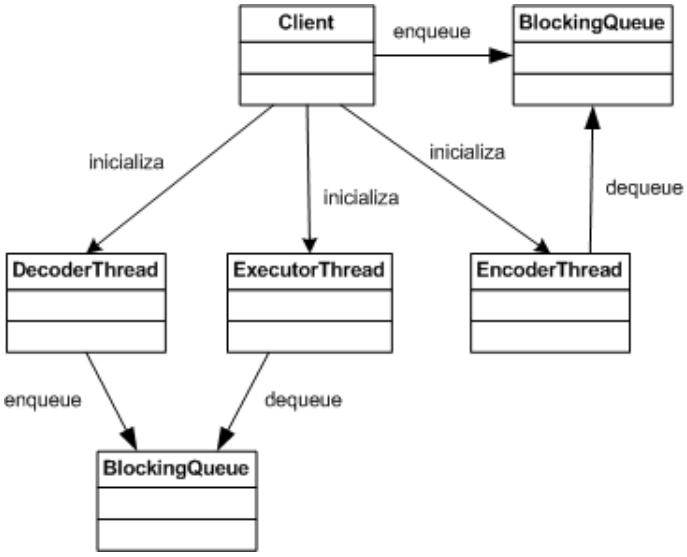

Figura 4.9 Diagrama de classes do núcleo do MCF

Antes de utilizar o *framework* para trocar mensagens pela rede, o mesmo deve estar conectado. Este procedimento é feito através de uma chamada ao método *invokeConnect* da classe *Client*. Este procedimento só é necessário ser feito uma vez, já que o *framework* mantém conexão permanente com o servidor. O *framework* também possui capacidade de notificar a aplicação em caso de queda de conexão.

Com a presença da *thread Client* fecha-se o ciclo de comunicação entre a aplicação e o servidor, feita através do *MCF*. Este processo é mostrado na figura 4.10. Pela figura, pode-se observar que o fluxo de dados segue tanto no sentido servidor-cliente quanto no sentido cliente-servidor.

Sentido servidor-cliente O fluxo de dados chega à aplicação através do **Decoder**, que é responsável por ler e decodificar as cadeias de bytes - de acordo com o protocolo binário da aplicação - e transformá-las em eventos (**Event**), que são mensagens bem tipadas (objetos) e podem ser facilmente interpretadas pela aplicação. Os eventos são enviados ao **Executor** que, a partir das informações recebidas, faz as devidas atualizações no estado da aplicação ou simplesmente notifica a aplicação sobre algum acontecimento.

Sentido cliente-servidor Quando a aplicação necessita solicitar ou enviar alguma informação ao servidor, ela o faz através do **Client**, que é a interface entre a aplicação e o *framework*. **Client** envia a solicitação - que no caso chamaremos de ação - encapsulada em mensagens do tipo **Action**, que são enviadas ao **Encoder**. Este, por sua vez, serializa os objetos **Action** em cadeias de bytes e as transmitem através do canal de saída da rede.

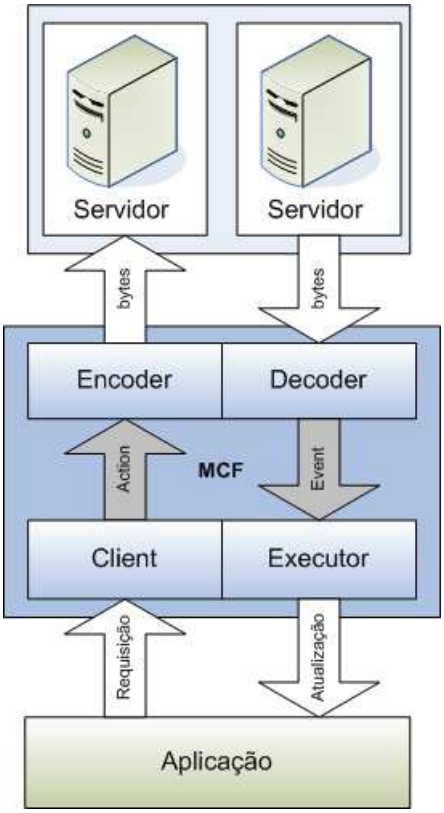

Figura 4.10 Visão geral da arquitetura do MCF

4.4 Considerações

O *MCF* foi implmentado utilizando-se a abordagem de caixa-branca, na qual o mesmo deve ser instanciado, para ser utilizado em um aplicação, através da deve herança e *overriding* de métodos ou implementação de interfaces. Isto significa que para o mesmo ser estendido, deve-se ter conhecimento de sua implementação, pelo menos no que diz respeito ao relacionamento entre as classes que o compõem. Porém, procuramos utilizar um modelo de programação abstrato que seja fácil de se entender, para que o aprendizado do *MCF* seja feito de forma rápida e fácil.

A implementação do *MCF* possui cerca de 400 linhas de código[5] divididos em 12 classes, que adicionam cerca de 12 kilobytes ao pacote da aplicação. Aspectos relacionados a tempo de desenvolvimento e *footprint* de memória não foram verificados sem o *framework* estar instanciado em uma aplicação, o que será feito nos capítulos seguintes.

[5]SLOCS, Medidas através do plugin Metrics, http://metrics.sourceforge.net/

CAPÍTULO 5

Estudo de Caso

Como foi dito anteriormente, durante o desenvolvimento deste trabalho implementamos o jogo OthelloME. Este jogo tem dois propósitos: validar a modelagem e implementação do *MCF*, e demonstrar como o desenvolvedor utiliza o *MCF* no desenvolvimento de uma aplicação conectada. Inicialmente apresentaremos uma visão geral das regras do jogo, em seguida mostraremos o método de instanciação do *MCF* para ser utilizado na implementação do jogo. Por fim, mostraremos os detalhes da implementação e analisaremos o impacto da utilização do *MCF* nesta implementação.

5.1 Regras

OthelloME é uma versão multiusuario para dispositivos móveis do clássico jogo de tabuleiro Othello [Ros05], também conhecido como Reversi. É um jogo de estratégia disputado por dois jogadores que se revezam em turnos em um tabuleiro quadrado em forma de grid de 8x8, com peças circulares que possuem dois lados distintos (preto e branco). A figura 5.1 mostra uma partida em andamento. O objetivo do jogo é ter a maioria de peças da própria cor no tabuleiro ao final do jogo. Conquista-se as peças do adversário ao cercá-las com peças da sua própria cor.

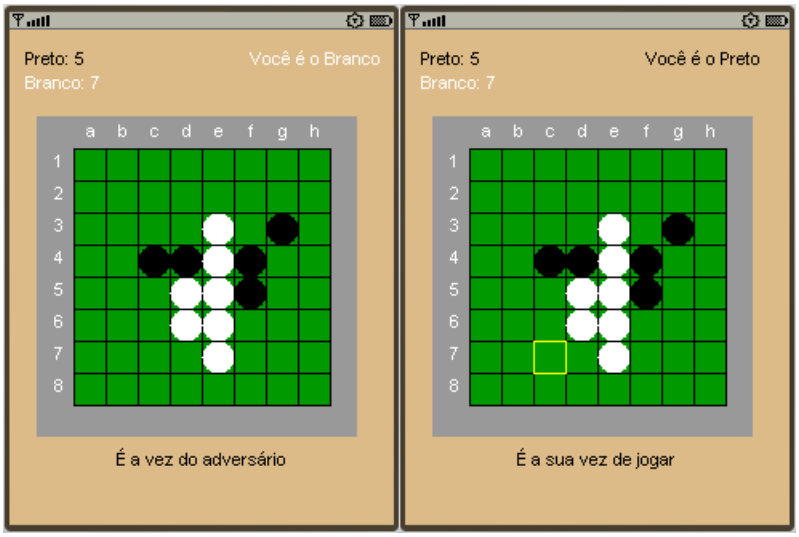

Figura 5.1 O jogo OthelloME

O jogo começa pelo jogador que possui as peças pretas e os turnos se alternam. O tabuleiro possui inicialmente peças pretas nas posições d5 e e4 e peças brancas nas posições d4 e e5. A cada turno o jogador deve fazer um movimento válido, que consiste em colocar uma peça de sua cor em uma posição vazia do tabuleiro de forma que consiga cercar peças do adversário. Este movimento se dá quando, entre duas peças alinhadas do jogador (vertical, horizontal ou diagonal), existe pelo menos uma peça do adversário e nenhum espaço vazio. Neste caso, as peças cercadas mudam de face e ficam da cor das do jogador. Caso o jogador não possua opção de movimento válido em seu turno, ele deve passar imediatamente o turno para o adversário.

5.2 Método de instanciação do MCF

Antes de utilizarmos o *MCF* devemos instanciá-lo para que o mesmo se adeque às mensagens utilizadas pelo OtheloME. Em outras palavras, devemos preencher o *framework* com código específico para fornecer as funcionalidades que o jogo requer. Em *frameworks* de caixa-branca como o *MCF*, este processo é realizado através da derivação de algumas classes abstratas através de herança, *overriding* de métodos e implementação de interfaces.

É de responsabilidade do *MCF* prover o fluxo de controle das mensagens, tanto no sentido da aplicação para o servidor quanto no sentido inverso. O código que implementarmos para estender o *MCF* será chamado implicitamente. Os pontos de extensão do *MCF* que precisamos implementar são divididos em três partes e serão descritos em seguida.

Envio de ações

Para cada mensagem que OtheloME pode enviar ao servidor (que chamamos de ação) devemos criar uma subclasse de *Action*, contendo atributos para cada um dos componentes do *payload* da mensagem, e sobrescrever o método *encode*, responsável por escrever estes atributos no fluxo de saída da rede. Com este procedimento podemos enviar as subclasses de *Action* para o servidor através do método *send*, da classe *Client*. Ao fazer isto, o método *encode* será chamado implicitamente, serializando os dados do objeto e os enviando pela rede.

Recebimento de eventos

Para cada evento que chega do servidor, devemos criar uma subclasse de *Event*, contendo atributos para cada parâmetro do *payload* do evento e sobrescrever o método *decode*. Neste método devemos incluir o código responsável por ler os dados do fluxo, preenchendo os atributos do evento, de acordo com a ordem estabelecida no protocolo. Em seguida devemos criar uma classe que implementa a interface *Factory*, que possui o método *create*. Este método é responsável por instanciar a implementação de *Event* responsável por tratar um determinado tipo de evento, onde o seu identificador é passado como parâmetro do método *create*. Abaixo está um exemplo de implementação desta classe.

```
public class EventFactory implements Factory {
```

```
public Object create(int eventType) {
    switch (eventType) {
    case MessageID.START:
        return new StartEvent();

    case MessageID.MOVE:
        return new MoveEvent();

        ...
    }
}
```

No momento em que chega um evento do servidor, através da *Factory* o *framework* obtém uma subclasse de *Event* responsável por tratá-lo, e o método *decode* desta classe é chamado automaticamente, preenchendo os atributos da classe e a enviando para ser processada.

Processamento de eventos

Cada evento gerado pelo *MCF* precisa ser processado para que o estado da aplicação seja atualizado. Por este motivo, para cada subclasse de *Event* deve existir um *Executor*, responsável por processar tal evento. *Executor* é uma interface que possui apenas o método *execute*, onde devemos adicionar o código para fazer tal processamento. De forma semelhante ao recebimento de eventos, devemos criar uma *Factory* para que o *framework* possa obter a instância que processa determinado evento.

5.3 Implementação

O primeiro passo a ser dado é a definição do protocolo binário que será utilizado na comunicação entre a aplicação cliente e o servidor. As mensagens devem ser definidas tanto para as ações quanto para os eventos. A tabela 5.1 mostra o protocolo binário que criamos para os eventos do jogo. O protocolo detalhado do jogo pode ser consultado em A.1.

Eventos	ID	Descrição
ERROR	0x00	Representa um erro no servidor.
START	0x01	Indica o início de uma partida.
TURN	0x02	Notifica o início de um turno.
PASS	0x03	Notifica que um jogador passou sua vez.
MOVE	0x04	Representa um movimento no jogo.
WIN	0x05	Indica que alguem venceu o jogo.
DRAW	0x06	Indica que o jogo foi empate.

Tabela 5.1 Protocolo de eventos do jogo

No jogo existe apenas uma ação, *MOVE*, que indica em que posição do tabuleiro o jogador está colocando sua peça. Em seguida, os identificadores das mensagens do protocolo do jogo são armazenados em uma interface chamada MessageID que contém constantes representando cada uma das mensagens, sejam elas eventos ou ações. Pode acontecer de uma ação ter um evento correspondente, no nosso caso a única ação do jogo, *MOVE*, possui um evento correspondente, nesse caso o mesmo identificador é utilizado para representar ambos.

Dado que já temos especificado o protocolo binário do jogo, implementaremos o primeiro ponto de extensão do *MCF*, para tanto devemos criar subclasses de *Action* e *Event* para representar cada mensagem do protocolo. Utilizamos a nomenclatura *XXXAction* e *XXXEvent* para cada ação ou evento que criarmos, onde XXX é o nome da mensagem do protocolo. O diagrama de classes que representa o modelo de mensagens do jogo é mostrado na figura 5.2. A ação *MOVE*, por exemplo, é mapeada em uma classe chamada *MoveAction*, que herda da classe *Action*. O código desta classe é mostrado em A.2.1.

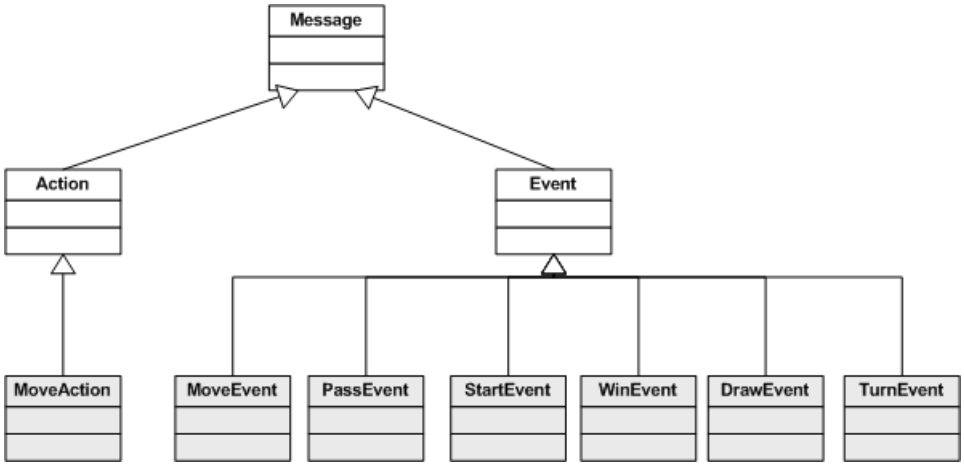

Figura 5.2 Diagrama de classes do modelo de mensagens do jogo

Observe que cada parâmetro que compõe uma mensagem é mapeado em um atributo da classe, no caso x e y constituem a posição em que o jogador irá colocar uma peça. O construtor de *MoveAction* simplesmente invoca o construtor de *Action* passando o byte que representa a mensagem *move* de acordo com o protocolo e configura as variáveis que representam os parâmetros da mensagem, a partir dos valores recebidos. Por fim o método *encode*, que recebe o fluxo de dados de saída e utiliza-o para escrever o tipo da ação e seus atributos na rede.

De maneira semelhante, para representar um evento de *move* que chega da rede, que pode ser tanto do jogador local quanto do adversário, criamos a classe *MoveEvent* que tem o código mostrado em A.2.2.

MoveEvent, que herda da classe *Event*, possui atributos que representam cada um dos elementos que constituem o *payload*[1] do evento. Neste caso piece, que representa se o movimento foi do jogador com a peça preta ou o da branca, e o par (x,y) que representa o movimento.

[1]É a parte variável de um protocolo binário, onde são passadas informações que são interpretadas de acordo com o cabeçalho da mensagem.

O construtor da classe invoca o construtor de *Event* indicando o tipo do evento, enquanto o método *decode*, que recebe o fluxo de entrade de dados da rede, lê os dados que vêm da rede na ordem especificada pelo protocolo preenchendo cada atributo da classe. No momento em que um evento é completamente decodificado, estes atributos podem ser acessados através de métodos de acesso (ex.*getPiece(), getX(), getY()*). De maneira similar, este processo é repetido para os outros tipos de eventos existentes no protocolo.

Por fim devemos criar as classes que implementam a interface *Executor*, responsáveis por realizar o processamento dos eventos e fazer as devidas atualizações no estado do jogo. Para cada *XXXEvent* que criamos devemos ter uma classe *XXXExecutor* que implementa o método *public void execute(Event event)* para processar o evento.

Na implementação de OthelloME criamos a classe *GameState*, que armazena informações relativas ao estado do jogo. Esta classe será atualizada todas as vezes que um novo evento chega ao jogo. Para tornar esta classe acessível a todas as classes que implementam *Executor*, criamos uma camada de indireção através da classe abstrata *ClientExecutor*, que implementa a interface *Executor* e contém uma referência à classe *GameState*. Então devemos agora estender a classe *ClientExecutor* para cada *XXXExecutor* e implementar o método abstrato *execute*, só que desta vez temos acesso ao estado do jogo para fazer as devidas atualizações.

Em A.2.3 é mostrada a implementação da classe *MoveExecutor*, responsável por processar um movimento feito por algum dos jogadores no jogo. As informações sobre o movimento estão encapsuladas em um *MoveEvent*.

Neste ponto, já somos capazes de utilizar o jogo para para trocar mensagens com o servidor. O envio de ações é feito através do método *send* da classe *Client*, lembrando que antes da primeira requisição deve-se invocar o método *invokeConnect()*, também da classe *Client* para que a conexão e os componentes do *framework* sejam inicializados. Em A.2.4 é mostrado como exemplo, o momento no jogo em que é enviada uma ação de nova jogada (*MoveAction*).

5.4 Considerações

A implementação de OthelloME foi feita desde o início com a utilização do *MCF* para prover conectividade, e desta forma, toda a funcionalidade de rede pôde ser realizada em poucas horas. O processo de extensão através de herança tende a ser bastante repetitivo, onde foram criadas 18 classes para prover o envio de ações e recebimento de eventos específicos do jogo. No entanto estas classes tendem a ser extremamente pequenas e especializadas, o que tora o código bem estruturado. Tais classes somaram no total cerca de 300 linhas de código, que possuem cerca de 11 *kilobytes* no pacote da aplicação. A soma das classes do núcleo do *MFC* junto com as classes específicas das mensagens de rede utilizadas no jogo, possuem cerca de 700 linhas de código.

CAPÍTULO 6

Otimizações

Premature optimization is the root of all evil.
—C. A. R. HOARE

No capítulo 4 apresentamos a modelagem e implementação do *MCF* com o objetivo de provermos uma solução reutilizável e extensível. Porém, não nos preocupamos com os aspectos relacionados ao desempenho de execução (tempo de processamento, consumo de memória, tamanho do código, etc.). Dependendo da complexidade da aplicação e da capacidade de memória e processamento dos dispositivos móveis o fator desempenho passa a ser de extrema importância.

Neste capítulo faremos uma análise cuidadosa dos elementos que compõem a implementação do *MCF* buscando identificar os pontos de melhoria em relação ao fator desempenho. Vale ressaltar que neste capítulo nosso foco principal não está nos aspectos relacionados a elegância e facilidade de utilização do *framework* e sim na obtenção do melhor desempenho em relação à alocação de memória, sincronização, processamento, etc.

A abordagem utilizada aqui será de analisar cada um dos dois sentidos em que as informações trafegam no *MCF* e também os mecanismos de sincronização utilizado pelos componentes em cada sentido. Em seguida será apresentada uma comparação entre a implementação do *MCF* mostrada no capítulo 4 e a otimizada.

6.1 Otimizações na codificação e envio de ações

O mecanismo de envio de ações ao servidor mostrado no capítulo anterior se dá através de orientação a eventos, onde objetos *Action* que representam os diversos tipos de ações são instanciados pelo desenvolvedor e enviados através do método *send(Action action)* da classe *Client*. Que por sua vez enfileira os objetos *Action* em uma fila de eventos para serem processados pela classe *EncoderThread*. Neste caso, o desenvolvedor deve criar subclasses de *Action* para cada ação utilizada pela aplicação e implementar o método *encode*, responsável por converter o estado da classe em um fluxo de bytes.

Embora seja esta uma solução elegante e extensível, há uma penalidade facilmente notada em relação ao consumo de memória, pois o mecanismo de envio faz uso intenso de criação de objetos. Em aplicações que enviam um grande número de ações em um curto intervalo de tempo (ex. Um jogo de nave onde cada disparo implica em uma ação ao servidor) será instanciado um elevado número de objetos neste mesmo intervalo, podendo consumir uma quantidade razoável de memória. Observe também que os objetos *Action* criados possuem um

curto tempo de vida, sendo utilizados apenas para transmitir informações entre duas *threads* do *framework* (Client e EncoderThread). Este processo de criação intensiva de objetos, que são descartados logo em seguida pode ocasionar dois problemas críticos em dispositivos com pouca memória disponível: estouro de memória, quando não há mais espaço suficiente para alocar os objetos, e maior consumo de processamento, pois o coletor de lixo passa a ser executado com maior freqüência.

Para solucionar este problema, é necessário modificar a forma pela qual as ações são enviadas ao servidor, de maneira que não se instancie um objeto para cada requisição. Devido ao fato dos objetos *Action* serem utilizados apenas em um escopo específico, podemos substituir a criação destes pela passagem direta das informações contidas no objeto, desta forma dando preferência à alocação estática ao invés da dinâmica [JDC+99]. Porém, a comunicação entre a classe *Client*, interface do *framework* com a aplicação, e a classe *EncoderThread*, responsável por enviar os dados ao servidor, é feita através e uma fila bloqueante (*BlockingQueue*) que está preparada para receber objetos *Action*, e não os tipos primitivos que representam a ação. Desta forma criamos uma nova entidade responsável por fazer a comunicação entre a classe *Client* e a classe *EncoderThread*.

6.1.1 A classe ParamHolder

ParamHolder é uma classe que armazena o conteúdo de uma ação a ser enviada pela rede. É constituída basicamente do identificador referente ao tipo da mensagem e dos parâmetros que serão enviados (*payload* da mensagem). Os dados são todos armazendos em um *buffer* de bytes, onde a conversão a partir dos diferentes tipos primitivos é feita automaticamente. Esta classe oferece um mecanismo de sincronização entre as *threads Client* e *EncoderThread* através do modelo produtor-consumidor [BA06] [T+01], também conhecido como *Bounded buffer* [H+98].

Este é um modelo clássico de sincronização entre processos, onde duas entidades, o produtor e o consumidor, compartilham um *buffer* de tamanho fixo. O produtor preenche o *buffer* com dados enquanto o consumidor remove os dados produzidos, um a um. Este processo acontece contínuamente, de maneira que o produtor nunca tentará adicionar novos dados enquanto o consumidor não remover os dados anteriores e o consumidor nunca tentará remover dados de um *buffer* vazio. Esta funcionalidade é fornecida através dos métodos *produce()*, *consume()* e *signal()*, que são mostrados na tabela 6.1.

Com a utilização da classe *ParamHolder*, os dados que compõem uma ação podem ser adicionados diretamente a um *buffer*, consumido pela classe *EncoderThread*, que os envia ao servidor. Desta forma economiza-se memória evitando a criação de objetos *Action*.

6.2 Otimizações no recebimento e processamento de eventos

O mecanismo de recebimento e processamento de eventos do servidor apresentado no capítulo 4 também utiliza uma abordagem orientada a eventos, onde as mensgens provenientes da rede são decodificadas pela classe *DecoderThread* e transformadas em objetos *Event* que são enviados a uma fila de eventos, sendo são processados pela classe *ExecutorThread* logo em

Método	Descrição
produce	Avisa à thread consumidora de que algo está sendo produzido
consume	Avisa à thread produtora de que algo está sendo consumido
signal	Notifica à thread que está esperando, de que a ação foi completa (produção ou consumo)
putXXX	adiciona dados ao buffer, onde XXX é cada um dos tipos primitivos (int, byte, etc.).
setMessageType	Indica o tipo de mensagem do protocolo que os dados do buffer representam

Tabela 6.1 Métodos da classe ParamHolder

seguida.

De forma similar ao mecanismo de codificação e envio de ações, a decodificação e processamento de eventos também apresenta pontos que podem ser otimizados visando menor consumo de memória e processamento. Observando a forma como os eventos são gerados para a aplicação nota-se que, para cada mensagem que chega do servidor, a classe *DecoderThread* cria um objeto *Event* que encapsula os dados da rede. Esta contínua criação de objetos pode causar os mesmos problemas levantados na seção anterior, caso o *framework* seja utilizado em aplicações que recebem uma grande quantidade de eventos em um curto espaço de tempo.

Observe também que o esquema de decodificação e de processamento de eventos é dividido entre duas *threads*, que se comunicam através de uma fila de eventos. A fila de eventos em questão é um objeto *BlockingQueue* e serve como ponto de sincronização entre as classes *DecoderThread* e *ExecutorThread*. A execução de código sincronizado (aquisição e liberação de *locks*) em Java é um processo custoso, que exige vários ciclos extra no processamento, podendo muitas vezes degradar o desempenho da aplicação. Para evitar esta sincronização desnecessária, removemos a fila de eventos e fundimos as classes *DecoderThread* e *ExecutorThread*. Desta forma, a decodificação e processamento de mensagens provenientes do servidor passam a ocorrer em uma mesma *thread* de execução, de forma Seqüencial. Assim, torna-se desnecessário o encapsulamento das mensagens do servidor em objetos *Event* e conseqüentemente não será mais necessário toda a abstração do modelo de mensagens, constituído pelas classes *Message*, *Action*, *Event* e as subclasses implementadas pelo desenvolvedor da aplicação.

A fusão entre as classes *DecoderThread* e *ExecutorThread* se dá da seguinte maneira:

- Todo o mecanismo de decodificação e processamento de eventos acontece na classe *DecoderThread*, enquanto a classe *ExecutorThread* é eliminada.
- A lógica processamento de cada evento é implementada na própria classe *DecoderThread*, ao invés de estar em cada subclasse de *Event*. Neste caso *DecoderThread* deve ter acesso às classes que modelam o estado da aplicação.
- O modelo de mensagens é eliminado e os pontos de extensão passam a ser feitos na própria classe *DecoderThread* através da implementação do método *decodeEvent(byte*

eventType, DataInput in). Neste método, é analisado o parâmetro *eventType* e a partir de seu tipo é chamado o método responsável por decodificar e processar o determinado evento.

Um trecho do método *decodeEvent* quando utilizado com o jogo OthelloME é mostrado a seguir.

```
private void decodeEvent(byte eventType, DataInput in) {
    if (eventType == MessageID.ERROR) {
        this.decodeErrorEvent(in);
        return;
    }

    if (eventType == MessageID.DRAW) {
        this.decodeDrawEvent(in);
        return;
    }

    if (eventType == MessageID.MOVE) {
        this.decodeMoveEvent(in);
        return;
    }

    // Continua para os eventos restantes
    ...

}
```

Cada método *decodeXXX* é responsável por ler os bytes da rede de acordo com o protocolo e também por atualizar o estado da aplicação através do processamento dos eventos. Abaixo é mostrado como exemplo, a implementação da decodificação e processamento do evento *MOVE* do jogo OthelloME.

```
private void decodeMoveEvent(DataInput buffer) {
    // Decodificação dos dados da rede, de acordo
    // com o protocolo de MOVE
    byte piece = input.readByte();
    byte x = input.readByte();
    byte y = input.readByte();

    // Atualização do estado da aplicação
    this.gameState.getBoard().update(piece,x, y);
}
```

6.3 Reestruturação da classe *Client*

Devido à eliminação do modelo de mensagens e a fusão entre as classes *DecoderThread* e *ExecutorThread* são necessárias algumas modificações na modelagem apresentada anteriormente. A ausência da classe *Action* inutiliza o método *send*. Lembre-se que o método *send* simplesmente adicionava o objeto *Action* a ser enviado, em uma *BlockingQueue* a ser consumida pelo *EncoderThread*. Com a substituição da sincronização através de filas bloqueantes por um modelo produtor-consumidor, através da classe *ParamHolder*, faz com que o mecanismo de envio precise ser feito através de tipos primitivos e não de objetos (*Action*, no caso).

Para que a passagem dos parâmetros que compõem uma ação seja feita através de tipos primitivos, a classe *Client* foi modificada com a adição de métodos de fachada. Desta forma a classe *Client* também passa a ser um ponto de extensão. Abaixo é mostrada a implementação do método *move*, responsável por enviar uma ação *MOVE* ao servidor.

```
public void move(byte x, byte y) {
    // Indica que está produzindo no buffer
    this.paramHolder.produce();

    // Configura o tipo da mensagem
    this.paramHolder.setMessageType(MessageID.MOVE);

    // Adiciona os parâmetros da ação
    // ao buffer
    this.paramHolder.putByte(x);
    this.paramHolder.putByte(y);

    // Sinaliza para que EncoderThread
    // consuma os dados do buffer
    this.paramHolder.signal();
}
```

Em OthelloME, a ação de enviar uma jogada no tabuleiro, que antes era realizada como mostrada em A.2.4, agora é realizada da seguinte maneira:

```
// Checa se é uma jogada possível
if (this.gameState.getBoard().check(
        this.gameState.getMyPiece(), this.selectorX,
        this.selectorY)) {

    // Envia um MoveAction
    this.client.move((byte)this.selectorX, (byte)this.selectorY);

}
```

6.4 Análise e comparação dos resultados

Após as otimizações mostradas neste capítulo serem realizadas, o *MCF* passou a contar com apenas quatro classes:

- *EncoderThread*: Responsável por enviar as ações através da rede.
- *DecoderThread*: Responsável por ler e processar os eventos provenientes do servidor.
- *Client*: Gerencia e sincroniza as classes *EncoderThread* e *DecoderThread*, como também as classes relacionadas à conexão com a rede. Possui métodos de fachada para que a aplicação possa enviar ações ao servidor.
- *ParamHolder*: Buffer de bytes utilizado para sincronizar a troca de dados entre *threads*.

Estas classes, junto com o código do jogo OthelloME, possuem 420 linhas de código, ocupando um total de 16,6 *kilobytes*. A utilização desta versão do *MCF*, como esperado, não apresenta boa modularidade e os *hot spots* são implementados diretamente nas classes *Client* e *DecoderThread*. Esta abordagem, por ser de mais baixo nível, exige que o desenvolvedor da aplicação tenha bom conhecimento da implementação do *framework*.

6.4.1 Consumo de memória

A principal variável analisada, o consumo de memória das duas versões do *MCF* foi medido através da ferramenta de *profiling* do *Sun Wireless Toolkit*[1]. Nesta medição, foram executadas partidas entre dois jogadores, cada um deles com uma versão diferente do *MCF*. A tabela 6.2 mostra a diferença entre o consumo de memória das duas implementações no início do jogo.

De acordo com estes dados observamos que a versão otimizada aloca 1044 *kilobytes* a menos, devido à fusão e remoção de algumas classes. Na tabela 6.3 é mostrado o comportamento do consumo de memória durante os dez primeiros turnos de uma partida.

Devido à constante criação de objetos feita pela primeira implementação do *MCF*, a variação na quantidade de memória alocada cresce a cada turno, à razão de 136 bytes. Este é o valor que conseguimos economizar durante a troca de mensagens com o servidor. É importante ressaltar que o jogo OthelloME utiliza poucas e pequenas mensagens para atualizar o estado do jogo. Em aplicações mais complexas, a economia de memória seria ainda mais perceptível.

Em resumo, a implementação de uma versão otimizada do *MCF*, onde evitamos a alocação contínua de objetos de curta duração, utilizou uma abordagem mais baixo nível, que é mais difícil de se entender o funcionamento e de ser personalizada de acordo com a aplicação se comparada à versão desenvolvida no capítulo 4. Entretanto esta abordagem, além de continuar possuindo vantagem sobre uma implementação de conectividade *ad-hoc* em termos de produtividade e reuso, também possui ganhos em relação a alocação de memória se comaparada à versão anterior.

[1] http://java.sun.com/products/sjwtoolkit/

Objeto	**Instâncias**	**Tamanho**
VM Internal	3	720
byte[]	1	-24
java.lang.Object[]	2	112
java.lang.Thread	-2	-56
java.util.Vector	2	48
othello.net.framework.ParamHolder	-1	-32
othello.net.framework.core.DecoderThread	1	20
othello.net.framework.core.EncoderThread	1	20
othello.net.framework.core.ExecutorThread	1	40
othello.net.framework.util.BlockingQueue	2	32
othello.net.framework.util.EventFactory	1	12
othello.net.framework.util.ExecutorFactory	1	40
othello.net.game.executor.DrawExecutor	1	16
othello.net.game.executor.ErrorExecutor	1	16
othello.net.game.executor.MoveExecutor	1	16
othello.net.game.executor.PassExecutor	1	16
othello.net.game.executor.StartExecutor	1	16
othello.net.game.executor.TurnExecutor	1	16
othello.net.game.executor.WinExecutor	1	16
Total		**1044**

Tabela 6.2 Diferença da alocação inicial de objetos entre as duas versões. (em unidades e bytes)

	0	**1**	**2**	**3**	**4**	**5**	**10**
Normal	62860	64544	65360	66176	66992	67808	75036
Otimizada	61816	62500	63180	63860	64540	65220	69532
Variação	1044	2044	2180	2316	2452	2588	5504

Tabela 6.3 Diferença do consumo de memória a cada turno. (em bytes)

CAPÍTULO 7

Conclusões

Neste trabalho apresentamos uma solução que visa tornar o desenvolvimento de aplicações móveis conectadas menos oneroso, através da utilização de um *framework*, responsável por abstrair aspectos de baixo nível relativos à conectividade, permitindo que o desenvolvedor direcione seus esforços para a implementação da lógica da aplicação. O *framework* em questão pode ser utilizado em aplicações que demandam garantia de entrega dos pacotes da rede e que toleram um certo atraso na entrega dos mesmos.

Na modelagem e implementação buscamos oferecer uma solução flexível que permita o reuso de forma prática, além de ser fácil de se estender, através da utilização de conceitos de programação orientada a eventos.

Além de buscarmos apresentar uma solução fácil de ser utilizada, também nos preocupamos com o desempenho do *framework*. Aspectos como utilização de memória e sincronização foram revistos, onde apresentamos pontos de otimização a serem feitos no *framework* caso o desempenho seja crítico para a aplicação.

A implementação foi estudada através de um estudo de caso, onde foi desenvolvido o jogo multiusuário móvel OthelloME. O estudo de caso foi útil para mostrarmos na prática a utilização do *framework*, além de nos ajudar a identificar os pontos de otimização que foram feitos posteriormente. Porém, algumas características do *framework* não foram exploradas da melhor forma através do estudo de caso, pois devido ao fato de se tratar de um jogo em turnos alternados, o jogo faz requisições sincronizadas com a resposta do servidor e em intervalo grandes, o que torna desnecessária a característica assíncrona do *framework*.

7.1 Trabalhos Futuros

Neste trabalho, englobamos uma determinada classe de aplicações conectadas, que podem ser beneficiadas com a utilização do *framework*. Contudo, há alguns pontos de melhoria e extensões que podem ser realizados no *framework*, tanto para complementar quanto para aperfeiçoar a solução existente.

Suporte a diferentes camadas de transporte e aplicação

A modelagem e implementação do *framework* contempla as aplicações móveis conectadas de propósito geral que não possuem tolerância à perda de pacotes, mas que toleram um certo atraso na entrega dos mesmos. Escolhemos a comunicação via *sockets TCP* para fazer o transporte dos dados na rede por ser o protocolo que se mostrou mais adequado aos requisitos definidos.

Porém, devido à utilização de uma arquitetura simples e modular é possível e também interessante a extensão do *framework* para fornecer suporte a outros protocolos de transporte e de aplicação, como:

- HTTP: Que é o protocolo com maior suporte nos dispositivos móveis e permite conectividade através de um modelo de requisição-resposta.
- UDP: Através do suporte a datagramas UDP pode-se avaliar a viabilidade de se desenvolver aplicações que utilizam serviço não orientado a conexão, que beneficia aplicações como *streaming* de áudio e vídeo.
- TLS: Aplicações que exigem um canal de comunicação seguro, podem ser beneficiadas com o suporte a este protocolo.

O suporte à conectividade via protocolos normalmente utilizados em arquiteturas *peer-to-peer* como *Bluetooth* e infravermelho e disponíveis através de *APIs* de Java ME, também pode ser estudado.

Geração automática de código otimizado

No capítulo 4 apresentamos a implementação do *framework* através de uma abordagem que tira proveito dos conceitos de orientação a objetos e da utilização de padrões de projeto conhecidos para proporcionar ao desenvolvedor um código de fácil de se entender e de ser estendido. Por outro lado, no capítulo 5 identificamos os pontos fracos da primeira abordagem, que estão relacionados principalmente ao consumo de memória, e apresentamos uma série de otimizações que podem ser feitas, produzindo uma versão do *framework* que consome menos memória, mas que por outro lado perde um pouco as caracteríscas de orientação a objetos e de reuso, por apresentar uma solução de mais baixo nível.

Uma possível linha de pesquisa seria o suporte à geração de código fonte ou binário da versão otimizada do *framework* a partir do código da versão inicial. Tal fato possibilitaria o aproveitamento das vantagens de cada uma das abordagens mostradas neste trabalho, pois o desenvolvedor poderia utilizar a primeira versão do *framework* em sua aplicação, que é mais fácil de ser estendida e re-utilizada, e posteriormente seria gerado automaticamente um código otimizado.

Generalização do *framework* para Java SE

Devido à constante evolução do poder de processamento dos dispotivos móveis, é cada vez mais próximo da realidade a utilização da versão padrão de Java (SE) nestes dispositivos. Java ME é considerada um subconjunto de Java SE. Porém as classes básicas de sua especificação, definidas no CLDC, são equivalentes às do Java SE versão 1.3 [Sun03], onde a versão mais atual é a 1.6, além disto, existem algumas classes que não fazem parte de Java SE e são específicas tanto de CLDC quanto de MIDP. Contudo, estas classes exclusivas ao Java ME são utilizadas na implementação do *framework* apenas na camada de transporte de dados na rede. Isto dá abertura a uma possível generalização, onde a camada de transporte de dados é adpatada à *API* de rede do Java SE.

Também é interessante a generalização completa do *framework* para Java SE, que explore os recursos mais recentes da linguagem como tipos genéricos, *APIs* de concorrência, coleções, enumerações, *annotations*, etc.

Estudos de caso mais detalhados

Em relação a estudos de caso da utilização do *MCF* há uma vasta gama de estudos a serem realizados. Devido a restrições de tempo, só foi possível analisar a implementação do *framework* através do jogo OthelloME, e mesmo assim este estudo não foi realizado da melhor maneira possível, pois a primeira versão do jogo já foi implementada com ajuda do *MCF*, onde o método ideal seria a implementação do jogo sem a utilização do *framework* e posteriormente com a utilização do mesmo, o que forneceria uma análise comparativa mais apurada.

Para uma validação do *framework* em relação a ganhos de produtividade e à curva de aprendizado, poderia-se submetê-lo a desenvolvedores de aplicações móveis conectadas que não tenham conhecimento prévio do *MCF*, para que os mesmos desenvolvessem aplicações com a ajuda do *framework*.

Finalmente, em relação à questão de desempenho pode-se, por exemplo, testar o *framework* com aplicações que demandam um grande número de mensagens diferentes no protocolo, e conseqüentemente, de eventos tratados pela aplicação, ou aplicações que utilizam com freqüencia chamadas assíncronas. Desta forma pode-se observar a diferença de comportamento entre as duas versões do *framework*.

APÊNDICE A

Detalhes do jogo OthelloME

Este apêndice especifica o protocolo binário detalhado do jogo OthelloME e mostra uma listagem de códigos úteis para o entendimento da implementação do mesmo.

A.1 Protocolo completo

A.1.1 Ações

Aqui estão descritas as possíveis ações a serem enviadas ao servidor.

MOVE

Representa um movimento do jogador, através da colocação de uma peça de sua cor no tabuleiro.

ID	**byte**	**byte**
0x04	x	y

- x: Posição horizontal da peça no tabuleiro. $1 \leq x \leq 8$
- y: Posição vertical da peça no tabuleiro. $1 \leq y \leq 8$.

A.1.2 Eventos

Aqui estão descritas os eventos recebidos pelo servidor do jogo.

ERROR

Representa um erro no estado de jogo do servidor.

ID	**byte**
0x00	type

- type: Tipo do erro. Em OthelloME só existe o erro 0x00, que indica uma jogada inválida.

START

Notifica o início de uma partida.

ID	byte
0x01	myPiece

- myPiece: A cor da peça do jogador, onde 0 representa preta e 1 representa branca.

TURN

Indica o início de um turno. É o momento de sincronização do placar do jogo com o servidor.

ID	byte	byte	byte
0x02	piece	blackCount	whiteCount

- piece: A peça que representa o jogador a qual pertence o turno. $0 \leq piece \leq 1$
- blackCount: Quantidade de peças pretas no tabuleiro. $0 \leq blackCount \leq 64$.
- whiteCount: Quantidade de peças brancas no tabuleiro. $0 \leq whiteCount \leq 64$.

PASS

Indica que algum jogador passou sua vez, por não poder fazer nenhum movimento válido no tabuleiro.

ID	byte
0x03	piece

- piece: A cor da peça que representa o jogador que passou a vez. $0 \leq piece \leq 1$

MOVE

Representa um movimento de algum dos dois jogadores no tabuleiro.

ID	byte	byte	byte
0x04	piece	x	y

- piece: A cor que representa o jogador que executou o movimento. $1 \leq piece \leq 8$
- x: Posição horizontal da peça no tabuleiro. $1 \leq x \leq 8$
- y: Posição vertical da peça no tabuleiro. $1 \leq y \leq 8$.

WIN

Indica que a partida terminou e alguem venceu.

ID	byte	byte	byte
0x05	winner	blackCount	whiteCount

- winner: A cor que representa a peça do jogador vencedor. $0 \leq winner \leq 1$
- blackCount: Quantidade de peças pretas no tabuleiro. $0 \leq blackCount \leq 64$.
- whiteCount: Quantidade de peças brancas no tabuleiro. $0 \leq whiteCount \leq 64$.

DRAW

Indica que o jogo terminou e foi empatado.

ID	byte	byte
0x06	blackCount	whiteCount

- blackCount: Quantidade de peças pretas no tabuleiro. $0 \leq blackCount \leq 64$.
- whiteCount: Quantidade de peças brancas no tabuleiro. $0 \leq whiteCount \leq 64$.

A.2 Listagem de códigos

Esta seção mostra apresenta trechos relevantes da implementação de OthelloME.

A.2.1 A classe MoveAction

```
package othello.net.game.action;

import java.io.DataOutput;
import java.io.IOException;

import othello.net.framework.Action;
import othello.net.game.MessageID;

public class MoveAction extends Action {

    private byte x;
    private byte y;

    public MoveAction(byte x, byte y) {
        super(MessageID.MOVE);
```

```
        this.x = x;
        this.y = y;
    }

    public void encode(DataOutput output) throws IOException {
        super.encode(output);
        output.writeByte(this.x);
        output.writeByte(this.y);
    }

}
```

A.2.2 A classe MoveEvent

```
package othello.net.game.event;

import java.io.DataInput;
import java.io.IOException;

import othello.net.framework.Event;
import othello.net.game.MessageID;

public class MoveEvent extends Event {

    private byte piece;
    private byte x;
    private byte y;

    public MoveEvent() {
        super(MessageID.MOVE);
    }

    public byte getPiece() {
        return this.piece;
    }

    public byte getX() {
        return this.x;
    }

    public byte getY() {
        return this.y;
    }
```

```
    public void decode(DataInput input) throws IOException {
        super.decode(input);
        this.piece = input.readByte();
        this.x = input.readByte();
        this.y = input.readByte();
    }

}
```

A.2.3 A classe MoveExecutor

```
package othello.net.game.executor;

import othello.net.framework.Event;
import othello.net.game.ClientExecutor;
import othello.net.game.event.MoveEvent;

public class MoveExecutor extends ClientExecutor {

    public void execute(Event event) {
        MoveEvent moveEvent = (MoveEvent) event;

        // Atualiza o tabuleiro de acordo com a jogada
        this.gameState.getBoard().update(moveEvent.getPiece(),
                moveEvent.getX(), moveEvent.getY());
    }
}
```

A.2.4 Envio de uma jogada através do *MCF*

```
// Checa se é uma jogada possível
if (this.gameState.getBoard().check(
    this.gameState.getMyPiece(), this.cellX, this.cellY)) {

    // Cria um MoveAction
    MoveAction moveAction = new MoveAction(
                    (byte) this.cellX, (byte) this.cellY);

    // Envia um MoveAction
    this.client.send(moveAction);
}
```

Referências Bibliográficas

[AM+06a] Carlos Eduardo Silva Andréa Menezes et al. Impacto da mobilidade no game design de 3mogs. *V Brazilian Symposium on Computer Games and Digital Entertainment*, November 2006. 2.1, 4.2

[AM+06b] Carlos Eduardo Silva Andréa Menezes et al. Uma plataforma para jogos móveis massivamente multiusuário. *V Brazilian Symposium on Computer Games and Digital Entertainment*, November 2006. 2.1, 4.2

[BA06] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, second edition, February 2006. 6.1.1

[Bou05] Bruno Costa Bourbon. Um framework para desenvolvimento de aplicativos em windows mobile. Trabalho de Graduação, August 2005. 1

[But] Greg Butler. Object-oriented application frameworks. Computer Science, University of Concordia, Montreal. 3.1

[EF+04] Elisabeth Freeman Eric Freeman et al. *Head First Design Patterns*. Head First. O'Reilly, first edition, October 2004. 4.3.2, 4.3.2, 4.3.2

[Fer06] Stephen Ferg. Event-driven programming: Introduction, tutorial, history, February 2006. 4.3.2

[G+95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, volume 1. Addison-Wesley Professional, 1995. 2.3, 3.2, 4.3.2, 4.3.2, 4.3.2

[H+98] Stephen J. Hartley et al. *Concurrent Programming: The Java Programming Language*. Oxford University Press, February 1998. 4.3.3, 6.1.1

[Hau97] Juha Hautamäki. A survey of frameworks. Technical report, University of Tampere, Finland, March 1997. 3

[JDC+99] Manish Gupta Jong-Deok Choi et al. Escape analysis for java. *OOPSLA'99*, 1999. 6.1

[Joh96] Ralph Johnson. Documenting frameworks using patterns. *OOPSLA*, 1996. 3.1

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10), October 1997. 3

[Nok03] Nokia. Optimizing the client/server communication for mobile applications. Technical report, Nokia, Inc., January 2003. Part 1. 4.3

[Nok04a] Nokia. Midp 2.0: Introduction to using sockets and datagrams. Technical report, Nokia Corporation, March 2004. 4.2.1

[Nok04b] Nokia. Multiplayer gaming performance over cellular networks. Technical report, Nokia Corporation, 2004. 2

[Pal03] Tommy Palm. The birth of the mobile mmog. *Gamasutra*, September 2003. 2.1, 4.2

[RJ97] Don Roberts and Ralph Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997. 3, 3, 3.2

[Ros05] Brian Rose. *Othello: A Minute to Learn... A Lifetime to Master*. 2005. 5.1

[RT+06] Stephan Buse Rajnish Tiwari et al. From electronic to mobile commerce: Technology convergence enables innovative business services. *Asia Pacific Tech Monitor*, October 2006. 2.1

[Sun03] Sun. Connected limited device configuration specification. Technical report, Sun Microsystems, Inc., 2003. 2.2, 7.1

[T+01] Andrew Tanenbaum et al. *Modern Operating Systems*. Prentice Hall, second edition, February 2001. 6.1.1

[WA+03] Francisco R. Cavalcanti Wellington Albano et al. Localização e segurança de dispositivos móveis entre redes cellular ip. *Boletim bimestral sobre tecnologia de redes - RNP*, July 2003. 2.1

[Wu04] Di Wu. Measurement based multiplayer gaming performance study over mobile networks. Master's thesis, IMIT, Royal Institute of Technology, 2004. 2, 2.1