Diego Mello da Silva

ALEA JACTA EST **- UM PROTÓTIPO DE JOGO DE ESTRATÉGIA INTERATIVO** ***MULTIPLAYER*** **PARA DISPOSITIVOS MÓVEIS**

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para obtenção do título de Bacharel

Orientador
Prof. Ricardo Martins de Abreu Silva

Lavras
Minas Gerais – Brasil
2003

Diego Mello da Silva

ALEA JACTA EST - UM PROTÓTIPO DE JOGO DE ESTRATÉGIA INTERATIVO _MULTIPLAYER_ PARA DISPOSITIVOS MÓVEIS

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para obtenção do título de Bacharel

Aprovada em *17 de Junho de 2003*

Prof. Guilherme Bastos Alvarenga

Prof. Ricardo Martins de Abreu Silva
(Orientador)

Lavras
Minas Gerais – Brasil

Sumário

1 Introdução **1**

2 Tecnologia **5**
2.1 Dispositivos Móveis Embarcados 5
2.2 Comunicação Wireless 6
2.3 Plataforma Java 2 8
2.3.1 As edições da plataforma Java 2 9

3 O jogo ***Alea Jacta Est*** **17**

4 Modelagem da Aplicação **25**
4.1 Interface 25
4.1.1 Iniciando a aplicação 25
4.1.2 Criando uma nova sessão de jogo 26
4.1.3 Entrando em uma sessão de jogo pré-existente 28
4.1.4 Interação durante o jogo 28
4.2 Arquitetura de Comunicação 32
4.3 Formato de Mensagens 38
4.4 Sincronização dos Turnos 42

5 Implementação **45**
5.1 O Estilo do Jogo 45
5.2 O Estado do Jogo 46
5.2.1 O Mapa 47
5.2.2 Os Elementos de Interação 49
5.2.3 Os Jogadores 52
5.2.4 O gerenciador de sessão 53
5.3 A lógica do jogo 54

5.3.1 Movimentando uma unidade militar 54
5.3.2 Abstendo-se de realizar uma ação para um elemento de interação 60
5.3.3 Criando uma fortificação 61
5.3.4 Criando uma unidade militar a partir de uma fortificação . 63

6 Considerações Finais 67
6.1 Comentários 67
6.2 Propostas de trabalhos futuros 70

A Formato de Mensagens 77
A.1 Mensagens de gerência de sessão de jogo 77
A.2 Mensagens de Ação 78

Lista de Figuras

2.1 A plataforma Java 2 e seus dispositivos alvo 10
2.2 Pilha de *software* típica da edição J2ME 15

3.1 Tipos de terreno do jogo *Alea Jacta Est* 18
3.2 Todos os movimentos possíveis para uma unidade militar 19
3.3 Unidades militares do jogo *Alea Jacta Est* 20
3.4 Sequência de criação de novas fortificações 21
3.5 Sequência de jogo envolvendo combate 23

4.1 Procedimento de início do jogo 26
4.2 Procedimento de criação de uma nova sessão do jogo 27
4.3 Procedimento para participar de uma sessão de jogo pré-existente 29
4.4 Dispositivo colorido padrão do *Wireless Toolkit* 2.0 com teclado `ITU-T keypad` 30
4.5 Sequência de criação de uma fortificação 31
4.6 Sequência de criação de novas unidades militares a partir de uma fortificação 33
4.7 Arquiteturas clássicas de comunicação utilizadas em jogos *multi-player* 33
4.8 Arquitetura cliente-servidor utilizada no protótipo *Alea Jacta Est* . 37
4.9 Formato das mensagens trocadas entre a porção cliente e servidor 39

5.1 Representação do mapa do jogo por trechos 49
5.2 Diagrama de estados para uma fortificação 52
5.3 Diagrama de estados para um jogador pertencente a uma sessão de jogo em execução 53
5.4 Tentativa de movimento inválido 55
5.5 Troca de mensagens para um movimento inválido 56

5.6 Tentativa de movimento válido 57
5.7 Troca de mensagens para um movimento válido 57
5.8 Tentativa de movimento que caracteriza uma ataque 59
5.9 Troca de mensagens para um ataque bem sucedido 59
5.10 Troca de mensagens para um ataque mal sucedido 60
5.11 Troca de mensagens para uma ação nula 61
5.12 Tentativa de criar uma fortificação a partir de um *sprite* não-*construtor* 62
5.13 Troca de mensagens para uma tentativa mal sucedida de criar uma fortificação 62
5.14 Criação de uma fortificação a partir de uma unidade *construtor* . . 63
5.15 Troca de mensagens envolvida na criação de uma fortificação . . . 64
5.16 Procedimento de criação de novas unidades militares 65
5.17 Procedimento de criação de novas unidades militares 66

Aos meus pais, por terem me dado a oportunidade de conhecer este mundo.

Agradecimentos

Aos professores, por manterem viva a transmissão do saber.

ALEA JACTA EST - UM PROTÓTIPO DE JOGO DE ESTRATÉGIA INTERATIVO *MULTIPLAYER* PARA DISPOSITIVOS MÓVEIS

O objetivo desse trabalho é apresentar um protótipo de jogo de estratégia interativo *multiplayer* denominado *Alea Jacta Est*. Esse jogo possui uma diferença substancial entre os jogos comumente encontrados em máquinas *desktop* pois foi desenvolvido para ser utilizado em dispositivos móveis como telefones celulares, PDAs e pagers. Este trabalho descreve a implementação do jogo apresentando informações sobre sua modelagem, interface de interação com o usuário, arquitetura de comunicação utilizada e controle do estado e lógica do jogo. Por fim será apresentado uma breve discussão sobre as dificuldades e facilidades encontradas no decorrer do trabalho, com propostas de melhoria capazes de torná-lo comercialmente viável.

ALEA JACTA EST - AN INTERATIVE AND STRATEGIC *MULTIPLAYER* GAME FOR MOBILE DEVICES

The purpose of this work is the presentation of a archetype strategic and interactive multiplayer game, named *Alea Jacta Est*. This game differs substantially from others games generally found on desktop machines, because it was developed to be used on mobile devices as cell phones, PDAs and pagers. This work describes the implementation of *Alea Jacta Est*, presenting informations about the modeling, user's interactive interface, communication architecture, control management and game's logic. To finalize, it's presented a discussion about the difficulties and easiness found during the development, showing improvement purposes capable to make it to be viable commercially.

Capítulo 1

Introdução

Desde o surgimento de computadores pessoais *desktop* na década de 70 as pessoas começaram a ver nestas máquinas um recurso de suporte as suas atividades cotidianas.

A partir da década de 80 o mercado de dispositivos embarcados alcançou os usuários comuns com aparelhos providos de uma CPU de propósito específico e funcionalidades domésticas como máquinas de lavar louça, cafeteiras elétricas, fornos microondas, videocassetes e outros aparelhos dessa categoria. A partir de então essas classes de dispositivos tornaram-se de consumo, sendo hoje indispensáveis ao conforto da vida cotidiana.

Da mesma forma, a partir da década de 90 surgiu a indústria de dispositivos móveis embarcados para comunicação *wireless* ou comunicação sem fio. Tratavam-se de dispositivos pequenos, alimentados a bateria, com baixa autonomia e inicialmente analógicos, como é o caso de telefones celulares. No entanto, pesquisas impulsionaram o desenvolvimento desses dispositivos utilizando tecnologia digital, que permitia algumas vantagens sobre a tecnologia de transmissão analógica, como por exemplo a possibilidade de transferir dados digitais sem a necessidade de um aparelho conversor analógico-digital além de estender a funcionalidade básica de comunicação por voz.

Com a popularização da internet também na década de 90 ocorreram mudanças substanciais nas relações de compra e venda com o surgimento do comércio eletrônico ou *e-commerce*, demonstrando o grande potencial que a rede pode oferecer devido a uma maior abrangência das lojas virtuais sobre o mercado consumidor que não depende mais dos limites físicos impostos pelo comércio tradicional. Outras modalidades de relações comerciais surgiram, como é o caso do *e-business* ou

negócio eletrônico que procura integrar todos os setores de uma empresa tradicional através de meios de comunicação como a internet, geralmente interligando clientes, funcionários, fornecedores e o governo de forma a obter melhor controle e rapidez nas transações comerciais e processos dentro da empresa.

Nos últimos anos o número de usuários de dispositivos de comunicação móvel tem crescido devido ao desenvolvimento de técnicas de múltiplo acesso como FDMA, TDMA e CDMA, de sistemas como AMPS e GSM e pela tendência de popularização dos serviços de banda larga. Conforme [1] estima-se que em 2007 nos EUA cerca de 60% da população estará fazendo uso de dispositivos móveis conectados à Internet. Da mesma forma estima-se que a China poderá ultrapassar a marca dos EUA tornando-se uma grande potência no setor devido ao crescimento do mercado de dispositivos sem fio. Para a União Européia estima-se que 75% das pessoas entre 15 e 50 anos carreguem consigo um dispositivo móvel.

Para a América Latina, pesquisas [1] tem demonstrado que o abismo existente entre esses países e a América do Norte tem diminuído, principalmente no que diz respeito a serviços de telefonia celular cuja participação subiu de 10% para 29%, demonstrando que o acesso *wireless* é bastante promissor na região. No Brasil, o número de aparelhos celulares cresceu de 17.605 no ano de 2000 para 35.090 no ano de 2002, podendo atingir a marca dos 41.500 até o ano de 2003, que indica um crescimento na popularização desses aparelhos.

Com base nestes dados é possível acreditar em um mercado em franca expansão dos serviços de comunicação móvel *wireless* aliado a tecnologias capazes de oferecer serviços de *e-commerce*, *e-business*, boletins eletrônicos, entretenimento e outros. Projeções feitas pela Nokia [2] indicaram que até o final de 2002 mais pessoas estariam acessando a internet por celulares do que por microcomputadores *desktop* convencionais no mundo. Embora isso não tenha ocorrido ainda, essa projeção criou boas espectativas para os desenvolvedores de jogos móveis e *publishers*. Destes serviços, um dos mais promissores é o entretenimento, como demonstra pesquisas divulgadas recentemente pela Datamonitor [3] que afirma que até 2005 cerca de 200 milhões de usuários de celulares estarão se divertindo com jogos móveis na Europa e Estados Unidos. Segundo o Shostek Group [4], para que isso se torne realidade é necessário que ocorram investimentos em novos sistemas, parcerias, redes e melhorias nos softwares para celulares, elementos estes que já devem estar disponíveis em 2005.

Apesar das diferenças entre o Brasil e os demais países que já se encontram adiantados em relação ao uso da Internet por meio de dispositivos sem fio, existem boas espectativas em relação à adoção dessa tecnologia pelos brasileiros. Segundo

o IBOPE [4], dos consumidores que têm a intenção de conectar-se a Internet 4,7 milhões devem fazê-lo por computadores *desktops*, enquanto que 1,9 milhão optaram por adquirir telefones celulares com suporte a WAP[1].

Atualmente várias tecnologias fornecem suporte ao desenvolvimento de aplicações para dispositivos móveis, como é o caso do protocolo WAP, BREW, da plataforma .NET e da edição *Java 2 Micro Edition* da plataforma Java 2. Diversos serviços podem ser oferecidos para usuários *wireless* por meio dessas tecnologias, que variam desde apresentação de conteúdo em WML, aplicações de *m-commerce*, serviços de localização, *e-mail* e entretenimento.

O objetivo deste trabalho foi implementar um protótipo de jogo de estratégia interativo *multiplayer* para dispositivos móveis. Tal protótipo foi desenvolvido utilizando a plataforma Java 2, sendo que a implementação da porção servidor do jogo utilizou a edição *Java 2 Standart Edition* enquanto que a porção cliente do jogo foi implementada utilizando a edição *Java 2 Micro Edition*. Detalhes sobre facilidades ou dificuldades de implementação de jogos para dispositivos móveis utilizando essa plataforma são dados em [5], que consiste em um trabalho de avaliação da tecnologia realizado em conjunto com o desenvolvimento desse protótipo.

Desenvolver jogos multiusuário para máquinas *desktop* é em geral uma tarefa complexa e desafiadora. O projeto de aplicações distribuídas deve tratar situações problemáticas que envolvem sincronização e concorrência. Dessa forma, projetar aplicações distribuídas para dispositivos móveis torna-se uma tarefa ainda mais desafiadora pois deve levar em conta gravíssimas restrições de processamento e memória, uma vez que dispositivos móveis como telefones celulares, PDAs e *pagers* são compactos e restritos no que diz respeito a recursos de *hardware* e *software*. Este trabalho apresenta algumas dificuldades encontradas no decorrer do desenvolvimento do protótipo, bem como soluções encontradas para contornar essas dificultades.

O capítulo 2 apresenta uma visão geral sobre a tecnologia de sistemas embarcados móveis, definindo o que são esses sistemas e como eles se comunicam, além de descrever uma das tecnologias que tornam possível a criação de aplicações para esses dispositivos por terceiros. No capítulo 3 é apresentado o jogo *Alea Jacta Est*, suas regras e objetivo. O total entendimento sobre o funcionamento das regras do jogo será necessário para compreender o trabalho, uma vez que o projeto do jogo foi orientado de forma a cumprir com a especificação criada. No capítulo 4 será

[1]WAP (*Wireless Application Protocol*) - consiste em uma pilha de protocolos de comunicação que tem como meta unir um servidor de aplicação a um dispositivo sem fio, permitindo acesso a Internet por meio deste.

apresentada a modelagem do jogo, descrevendo aspectos como interação dos jogadores, arquitetura de comunicação empregada, formato das mensagens trocadas e esquema de sincronização dos turnos de jogo. No capítulo 5 será apresentada uma descrição detalhada sobre a implementação do jogo, com ênfase no uso da interface para interação, da comunicação entre a porção cliente e servidor gerada em resposta a uma ação do usuário e da lógica empregada para manter e controlar a consistência do jogo. Por fim será apresentada uma conclusão do trabalho no capítulo 6, enfatizando as dificuldades encontradas no decorrer do projeto, as facilidades oferecidas pela tecnologia empregada na implementação do jogo e propostas para trabalhos futuros visando melhorar o protótipo.

Capítulo 2

Tecnologia

Este capítulo apresenta uma visão geral sobre o mundo dos dispositivos embarcados móveis. A primeira seção explica brevemente o que vem a ser dispositivos embarcados e em qual contexto esses encontram-se inseridos. Em seguida será apresentada uma visão geral sobre comunicação sem fio, que é a base para a construção de dispositivos embarcados móveis. Por fim um breve descrição da plataforma Java 2 é dada na seção 2.3. Essa plataforma consiste em uma das tecnologias possíveis de serem empregadas para o desenvolvimento de aplicações para dispositivos móveis, e é a tecnologia utilizada nesse trabalho.

2.1 Dispositivos Móveis Embarcados

A tecnologia dos computadores está presente em todos os lugares na nossa vida diária sobre a forma de produtos baseados em comunicação *wireless*. Ao olhar a nossa volta observamos um ambiente repleto de exemplos desses tipos de dispositivos, que se manifestam principalmente na forma de telefones celulares para comunicação móvel - capazes de acessar informações remotamente - ou na forma de cartões para telefones celulares, que armazenam informações a respeito de um usuário, como por exemplo sua quota de serviço disponível.

Dispositivos móveis embarcados consistem em uma classe de dispositivos pequenos que cabem na palma da mão ou no bolso de um usuário e que possuem um pequeno computador dedicado dentro deles. Nessa categoria encontram-se dispositivos como telefones celulares e *pagers*. Existe outra classe de dispositivos embarcados que são fixos conhecidos como dispositivos de consumo que compreendem aparelhos maiores e mais robustos que os dispositivos móveis. Exemplos

de dispositivos que pertencem a esta classe são máquinas de lavar, fornos micro-ondas, sistemas de navegação automobilística e terminais de ponto de venda.

A principal diferença entre esses dispositivos e os sistemas computacionais convencionais está no fato de que os sistemas convencionais são capazes de executar uma extensa faixa de aplicações de propósito geral, enquanto que os dispositivos embarcados são limitados ao seu domínio de aplicação, executando aplicações muito específicas. Outra característica desses dispositivos é que a cerca de 5 anos atrás sua arquitetura era fechada o suficiente para que a comunidade em geral não fosse capaz de desenvolver ou instalar novos programas para os mesmos, característica essa que tem mudado devido ao esforço de fabricantes de sistemas embarcados e empresas de software em abolir esse modelo a exemplo da indústria de software para PC's [6].

Algumas tecnologias permitem desenvolver aplicações para essa classe de dispositivos, aproveitando as vantagens de conectividade fornecida por eles aliado a aplicações que podem oferecer serviços como *e-mail* e acesso a banco de dados remoto de forma semelhante a computadores *desktop*. Dentre essas tecnologias, uma das mais utilizadas nos dias atuais tem sido a plataforma Java 2 e a edição Java 2 Micro Edition da Sun Microsystems que serão referenciadas na seção 2.3. A plataforma J2ME tem tido uma grande aceitação por parte dos desenvolvedores devido a sua característica de portabilidade que permite o desenvolvimento de aplicações capazes de executar em diferentes dispositivos, além do interesse dos fabricantes de dispositivos móveis embarcados em darem suporte a essa tecnologia como é o caso da Nokia, da Motorola, da Siemens e outras grandes empresas do ramo.

2.2 Comunicação Wireless

Comunicação *wireless* define o tipo de comunicação sem fio entre dispositivos. Engloba todo e qualquer tipo de comunicação onde não exista um meio físico capaz de transportar as informações entre dispositivos, sejam eles móveis ou fixos. São muitos os exemplos de dispositivos embarcados que utilizam esse tipo de comunicação, variando desde controles remotos de aparelhos televisores, brinquedos eletrônicos como *walk-talkies* ou carrinhos de controle remoto, alarmes de carros operados a distância por meio de chaveiros, controles remotos para abertura de portões em uma residência, e aparelhos celulares, *pagers* e PDA's capazes de transmitir voz e dados em formato digital por meio de uma rede *wireless* conectada a uma empresa ou a própria *internet*.

Segundo [7], a informação pode ser transmitida através dos dispositivos de várias formas possíveis, mas as mais comuns são o uso do infra-vermelho e das ondas de rádio-frequência. A comunicação ocorre devido a presença de transmissores e/ou receptores embutidos nos dispositivos embarcados, onde a presença dos mesmos classificam esses aparelhos em *one-way*, que apresentam apenas um transmissor ou receptor ou então *two-way*, que apresentam tanto um transmissor quanto um receptor no aparelho permitindo comunicação bidirecional.

Dispositivos móveis de comunicação *wireless* utilizam geralmente o espectro de rádio para comunicação. O espectro de rádio compreende uma larga faixa de radiação eletromagnética de baixa frequência, onde as ondas operam na faixa de 3 KHz até 300 GHz. Para se ter idéia de escala, os comprimentos de onda correspondentes a essas frequências variam de 300 metros a 3 centímetros. Em geral o espectro de rádio é dividido em canais ocupando cerca de 30 KHz de banda cada um, que servem tanto para transmitir sinais de comunicação quanto para transmitir sinais de controle para os dispositivos. Pelos canais de comunicação trafegam dados e voz, enquanto que pelos canais de controle trafegam informações de comandos e sinalização de entrada e saída, por exemplo.

O uso das frequências de rádio para esse tipo de comunicação deve-se à característica dos comprimentos de onda maiores penetrarem obstruções como por exemplo florestas, montanhas e cidades sem distorcer significativamente o sinal enviado, além de viajar grandes distâncias até serem transmitidas até satélites e refletidas para qualquer lugar do continente. Além disso ondas com baixa frequência são caracterizadas por possuírem baixa energia, podendo incluive causar danos à saúde humana.

As primeiras pesquisas em comunicação *wireless* na América foram realizadas pela AT&T quando lançou seu sistema AMPS (*Advanced Mobile Phone System*) por volta de 1980. Devido a sua característica de operar sobre sinais analógicos, o sistema AMPS é mais adequado para uso em voz do que em dados funcionando com conexões dedicadas. O sistema AMPS faz uso da técnica de divisão de canais FDMA (*Frequence Division Multiple Access*), que consiste em dividir o espectro concedido para a prestadora de serviços em pequenos canais suficientes para transmitir voz, não fazendo uso eficiente da largura de banda disponível.

Para resolver esse problema surgiram técnicas de multiplexação de canais como é o caso do TDMA (*Time Division Muliple Access*) e do CDMA (*Code Division Multiple Acess*). Na técnica de multiplexação TDMA cada canal de voz é dividido em *slots* de tempo, permitindo que até três usuários ocupem o mesmo canal de comunicação triplicando a capacidade da rede em relação ao sistema AMPS.

A técnica empregada pela multiplexação CDMA é substancialmente diferente da empregada pela multiplexação TDMA, consistindo em permitir que todos os usuários compartilhem a mesma frequência do espectro de rádio ao mesmo tempo. Para que isso se torne possível, os sinais de cada usuário são distinguidos por uma única seqüência de código. Cada comunicação de voz ou dados é quebrada em pequenos pedaços, e a cada pedaço é dado um código de identificação. Quando o sinal é recebido ele pode ser extraído e reconstruído conhecendo a sequência de código que foi enviada, uma vez que o *software* do recebimento é capaz de isolar e remontar a mensagem original.

O sistema GSM foi criado como padrão na Europa pelo *European Telecommunications Standards Institute* (ETSI) devido a proximidade entre as nações que compõem o continente, consistindo em um conjunto de padrões necessários para criar uma rede *wireless* capaz de cobrir toda a área conhecida como União Européia de forma a garantir interoperabilidade sobre a rede além dos limites nacionais. São baseados na tecnologia TDMA, e utilizam um SIM Card (*Subscriber Indentify Module*) que consiste em um cartão com um *chip* contendo informações sobre a identidade e autenticação do assinante. Tais cartões são altamente portáveis entre dispositivos GSM, de forma que o usuário remove seu SIM Card de um aparelho na Espanha e ao colocá-lo em outro aparelho GSM na França ele ainda é identificado, sendo que a rede reconhece-o como único usuário independente de qualquer aparelho GSM.

O desenvolvimento da tecnologia de transmissão *wireless*, o projeto e construção de dispositivos embarcados móveis com capacidade de processamento maiores que os seus antepassados, a abertura da arquitetura desses dispositivos pelos próprios fabricantes e a alta aceitação por parte dos usuários dos serviços móveis oferecidos através da *internet* criam um ambiente adequado para a pesquisa e desenvolvimento de novas tecnologias e aplicações direcionadas a esse segmento do mercado. Nesse contexto apresentaremos nas próximas seções a plataforma Java 2. Tal tecnologia torna possível o desenvolvimento de aplicações para dispositivos móveis embarcados e fácil integração com outras tecnologias já existentes, tornando-se a escolha adequada para implementação de serviços em ambientes corporativos e entretenimento em geral.

2.3 Plataforma Java 2

O propósito desse trabalho foi implementar um jogo interativo distribuído para dispositivos móveis. Existem atualmente diversas tecnologias utilizadas para o de-

senvolvimento de aplicações para dispositivos móveis, como é o caso do protocolo WAP, do BREW, da plataforma .NET e da edição J2ME da plataforma Java 2.

A plataforma Java 2 foi a tecnologia escolhida para a implementação do protótipo do jogo devido a diversos fatores, sendo que os principais são:

- portabilidade oferecida pela linguagem Java, que facilita a execução da porção servidor em qualquer *hardware* ou sistema operacional com suporte ao ambiente de execução do Java;

- economia de tempo no aprendizado de uma única linguagem de programação. Embora a porção cliente e servidor tenha sido implementados com pacotes de duas edições diferentes da plataforma Java 2, algumas classes são comuns às duas;

- suporte a *multithreading*, sem o qual seria difícil implementar diversas sessões de jogo executando concorrentemente no mesmo *host*;

- suporte ao *garbage collector*, que encoraja o desenvolvimento de aplicações bastante robustas sem a preocupação do desenvolvedor com a gerência de memória;

- grande quantidade de classes utilitários (vetor, listas, tabelas *hash*) intrínsecas à própria linguagem que a tornam completa, sem a necessidade de utilizar classes e pacotes de terceiros.

- facilidade de integração entre a porção cliente e servidor do jogo devido ao fato de pertencerem a mesma plataforma.

Um entendimento mais completo sobre a linguagem e características que ela suporta pode ser encontrado em [8].

Atualmente a plataforma Java 2 está dividida em três edições, cada uma direcionada a diferentes tipos de aplicação. As edições que compõem a plataforma Java 2 são as edições *Enterprise Edition*, *Standard Edition* e *Micro Edition*, que serão descritas abaixo.

2.3.1 As edições da plataforma Java 2

Atualmente a plataforma Java 2 é dividida em três edições que possuem enfoques diferentes no desenvolvimento de aplicações. Isso permite um melhor direcionamento das necessidades do desenvolvedor além da evolução de cada edição da

plataforma de forma totalmente independente das demais. As três edições que compreendem a plataforma Java 2 e seus dispositivos alvo estão ilustradas na Figura 2.1 e descritas abaixo:

Standard Edition (J2SE) [9], cujo alvo são as aplicações *desktop* convencionais;

Enterprise Edition (J2EE) [10], com ênfase no desenvolvimento de aplicações *server-side* para empresas com necessidade de servir seus consumidores, fornecedores e empregados com soluções sólidas e completas de negócios pela Internet;

Micro Edition (J2ME) [11], orientada a dispositivos embarcados e de consumo, tais como telefones celulares e PDA's (*Personal Digital Assistants*) que não são capazes de suportar uma aplicação J2SE completa. Representa um retorno às origens do Java (*The Green Project*).

Cada edição Java define um conjunto de tecnologias e ferramentas que podem ser usadas com um produto particular, tais como máquinas virtuais Java que abrangem um grande conjunto de dispositivos computacionais, bibliotecas e APIs especializadas para cada tipo de dispositivo computacional e ferramentas para expansão e configuração de um dispositivo.

Figura 2.1: A plataforma Java 2 e seus dispositivos alvo

Este trabalho utiliza as edições J2SE e J2ME para o desenvolvimento do protótipo. A porção servidor foi implementada em J2SE principalmente pelo fato de que essa edição é a mais adequada para desenvolvimento de aplicações para computadores *desktop*. Seus pacotes `java.io` e `java.net` contêm as classes básicas utilizadas para comunicação (*sockets* e *streams*). Além disso ela permite que mesmo um simples computador *desktop* funcione com servidor de jogo, bastanto para isso ter instalado o ambiente de execução da plataforma Java. Detalhes sobre a implementação de aplicações utilizando essa edição podem ser encontrados em [12].

Por outro lado, a porção cliente foi implementada utilizando a edição J2ME. Essa edição é composta por combinações de pacotes e bibliotecas de classe orientadas a diferentes sistemas embarcados. Isso reflete a e grande heterogeneidade de dispositivos que a plataforma é capaz de fornecer suporte, dispositivos estes que variam de aparelhos de consumo (lavadoras, fornos micro-ondas e outros), aparelhos de localização via-satélite até aparelhos de comunicação móvel (telefones celulares, PDAs e *pagers*).

A edição J2ME atualmente é composta por um conjunto de *perfis* e *configurações*. Um perfil é uma especificação de API's Java definidas pela indústria que são utilizadas por fabricantes e desenvolvedores em tipos específicos de dispositivos. Uma configuração consiste em uma máquina virtual Java e um conjunto mínimo de bibliotecas de classe básica e API's. Ela especifica um ambiente de execução geral para dispositivos embarcados e de consumo, atuando como uma plataforma Java no dispositivo capaz de fornecer ao fabricante do dispositivo ou a um provedor de conteúdo tudo o que ele espera estar disponível em todos os dispositivos de uma mesma categoria. Detalhes sobre a implementação de aplicações utilizando J2ME podem ser encontrados em [13], [14] e [15]

Tanto os perfis quanto as configurações em J2ME são definidas pela *Java Community Process* [1] (JCP) que consiste em um grupo de especialistas compostos por companhias que representam a indústria de dispositivos móveis e embarcados.

Ao combinar as API's fornecidas para um determinado perfil com as API's e máquina virtual oferecidas para uma configuração o desenvolvedor tem tudo o que precisa para implementar uma aplicação Java orientada a uma determinada classe de dispositivos uma vez que os fabricantes desses dispositivos garantem a portabilidade para aplicações desenvolvidas por terceiros em J2ME que utilizam o mesmo perfil e a mesma configuração.

[1] `http://www.jcp.org/`

Perfil

Enquanto a portabilidade de aplicações é o maior benefício da tecnologia Java nos ambientes *desktop* e *enterprise*, ela torna-se um elemento crítico no desenvolvimento de aplicações para dispositivos móveis e embarcados. Em muitos casos esses dispositivos possuem diferenças substanciais em tamanho de memória, capacidades de interface e conectividade, sendo impossível encontrar uma única solução capaz de contemplar todos os dispositivos por estarem em diferentes domínios de aplicação como é o caso de telefones celulares, máquinas de lavar e brinquedos eletrônicos. Da mesma forma, muitas vezes é indesejável a portabilidade entre tais dispositivos, de forma que o consumidor não deseja que uma aplicação de serviços automotivos funcione em sua máquina de lavar.

Existem ainda razões econômicas para manter esses dispositivos em classes diferentes uma vez que dispositivos de consumo competem entre si em custo e conveniência, onde tais fatores traduzem-se em limitações no tamanho físico, capacidade de processamento, tamanho de memória e consumo de energia para o caso de dispositivos alimentados por bateria.

A solução encontrada pela Sun Microsystems para dar suporte a plataforma Java entre os diferentes segmentos foi definir o conceito de perfil. Um perfil fornece um conjunto de ferramentas para o desenvolvimento de aplicações para diferentes tipos de dispositivos como por exemplo *pagers*, telefones celulares, máquinas de lavar, fornos microondas e outros. Um perfil também pode ser criado para suportar um grupo de aplicações que podem ser hospedadas em diferentes tipos de dispositivos como é o caso de aplicações de *home-banking* através da criação de um perfil separado para esse tipo de aplicação e garantia de suporte para cada dispositivo alvo.

Pode acontecer de um dado dispositivo suportar diversos perfis. O fabricante do dispositivo decide quais perfis seu dispositivo irá suportar, implementando todas as características necessárias dos perfis escolhidos. Cada aplicação é escrita para um perfil particular de forma a rodar em qualquer dispositivo que possua suporte a esse perfil.

Em nível de implementação, um perfil é definido como uma coleção de API's Java e bibliotecas de classe que se situam no topo de uma configuração e que fornecem capacidades específicas ao domínio de aplicação para dispositivos de um determinado segmento do mercado. As especificações das API's em J2ME são baseadas em J2SE com algumas modificações para melhor fornecer suporte às necessidades de cada produto.

Para implementação desse protótipo foi utilizado o perfil *Mobile Information*

Device Profile (MIDP) [16]. O MIDP especifica um perfil direcionado para o desenvolvimento de aplicações em dispositivos móveis com conexão *wireless*. Esses dispositivos possuem ainda *display* reduzidos, dispositivos de entrada e armazenamento local limitados, dependente de baterias como fonte de energia, e com uma CPU de propósito específico. Fornece uma plataforma padrão para dispositivos de informações móveis pequenos com recursos limitados caracterizados por:

- 512k de memória total (entre RAM e ROM) disponível para as bibliotecas e ambiente de execução Java;
- conectividade para algum tipo de rede *wireless* com largura de banda limitada;
- consumo de energia limitado, tipicamente fornecido por bateria;
- diferentes graus de sofisticação na interface com o usuário.

Outros perfis encontram-se atualmente em desenvolvimento. Cada um deles é direcionado a dispositivos com diferentes funcionalidades.

Configuração

Uma configuração define a plataforma Java para um grupo de dispositivos com características similares no que diz respeito às limitações do mesmo. Uma configuração em J2ME basicamente:

- especifica as características da linguagem Java suportadas;
- especifica as características da máquina virtual Java suportada;
- especifica as bibliotecas de classe básicas e API's suportadas.

Existem atualmente duas configurações para a plataforma J2ME, denominadas CDC e CLDC cujos dispositivos alvo são melhor especificados abaixo. É muito difícil definir o limite entre cada classe de dispositivo, que torna-se menor a cada dia devido às convergências tecnológicas entre os computadores, as telecomunicações, o projeto de dispositivos embarcados e a indústria de entretenimento. Dessa forma, a linha que divide essas classes é definida mais em razão da quantidade de memória, consumo de bateria e tamanho de tela do dispositivo em questão do que em razão da funcionalidade ou conectividade do mesmo.

A *Connected Device Configuration* (CDC) [17] foi desenvolvida para dispositivos com quantidades relativamente maiores de memória, fornecendo em geral dois megabytes ou mais para a plataforma Java. É composta pela máquina virtual CVM e pelo *Foundation Profile*, perfil este que contêm todas as bibliotecas de classe e API's projetadas para dispositivos embarcados de consumo (máquinas de lavar, fornos microondas e afins).

Os dispositivos alvo dessa configuração geralmente possuem mais memória para o ambiente Java, e em geral requerem toda a funcionalidade da máquina virtual *Compact Virtual Machine* (CVM). Tipicamente contêm algum tipo de conectividade *wireless* com largura de banda limitada (frequentemente 9600 bps ou menos), como por exemplo pagers, PDA's, terminais de ponto de venda e sistemas de navegação automobilística. Em geral são fixos, e muito frequentemente utilizam o protocolo TCP/IP para comunicação em rede.

A *Connected Limited Device Configuration* (CLDC) [18] é uma configuração orientada a dispositivos de informação móveis cuja memória varia na faixa de 160Kb a 512Kb. É composta pela máquina virtual *Kilobyte Virtual Machine* (KVM) e um conjunto de classes que podem ser usadas por uma grande variedade desses dispositivos.

Uma importante característica da configuração CLDC é que ela introduz o *Generic Connection Framework* (GCF), que consiste em uma hierarquia de classes e *interfaces* para criar conexões do tipo HTTP, datagrama ou *stream* e realizar entrada e saída. Ela é implementada pelo pacote `javax.microedition.io` e fornece conexão para dispositivos que não possuem memória suficiente para utilizar pacotes maiores como `java.net` ou `java.io`.

A máquina KVM foi projetada para ser utilizada em dispositivos pequenos, com recursos limitados de memória e conectividade com redes *wireless*. Estes dispositivos contêm geralmente processadores de 16 ou 32 bits, e um mínimo de memória total de 128Kb. Além disso, possuem interface com o usuário muito simples se comparadas com sistemas computacionais *desktop* além de largura de banda reduzida, onde a comunicação de redes frequentemente não se baseia no protocolo TCP/IP.

Muitos dos perfis disponíveis para J2ME baseiam-se também em outros perfis de forma que as novas funcionalidades complementam as já existentes, criando aplicações mais completas para uma determinada configuração. Ao combinar um ou mais perfis com uma configuração da edição J2ME cria-se a chamada pilha de *software*. Tal pilha tem uma configuração como a base para o desenvolvimento da aplicação. No caso do desenvolvimento do protótipo a configuração escolhida foi

a CLDC por cumprir com restrições impostas pelos dispositivos móveis alvo da aplicação. Sobre uma configuração utiliza-se um dos perfis que ela suporta. Para o *Alea Jacta Est* o perfil escolhido foi o MIDP, pois é orientado a funcionalidade de aparelhos móveis. A pilha de *software* final necessária para a implementação do protótipo *Alea Jacta Est* é ilustrada na Figura 2.2.

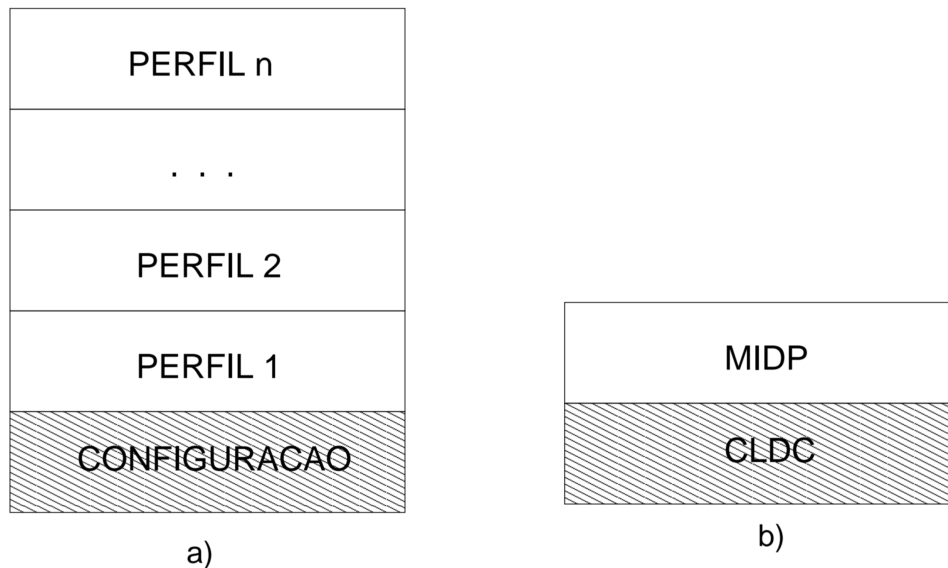

Figura 2.2: Pilha de *software* típica da edição J2ME. a) pilha de *software* genérica composta por vários perfis que fornecem funcionalidades variadas ao dispositivo. b) pilha de software do protótipo *Alea Jacta Est*.

Capítulo 3

O jogo *Alea Jacta Est*

Nesse capítulo será apresentada a especificação de um jogo de estratégia interativo *multiplayer* denominado *Alea Jacta Est* que será empregado como estudo de caso de implementação de jogos para dispositivos móveis. Tal especificação consiste na formalização das regras e objetivos do jogo, e será apresentada como uma descrição alto-nível do jogo, não importando-se com detalhes de implementação que serão melhor explicados nos próximos capítulos deste documento.

O nome do jogo é uma referência para uma célebre frase do general romano Caio Júlio César (100-44 a.C.), que traduzida do latim significa "A sorte está lançada". Conta a história que o general Júlio César proferiu tal frase momentos antes de marchar em direção a capital romana para enfrentar as tropas do general Pompeu, outro membro do triunvirato, mesmo estando suas tropas em minoria [19]. Tal episódio da história serviu como inspiração para a temática do jogo, que é repleto de referências ao império romano.

O jogo *Alea Jacta Est* é um jogo onde muitos participantes podem jogar ao mesmo tempo. Cada jogador representa um general romano sedento para expandir seu domínio pelo mundo, representado por um mapa. Para tal este conta com o apoio de seus exércitos, representados por unidades militares romanas da antiguidade e de fortificações construídas por ele em posições estratégicas do mapa. Cada fortificação permite a construção de novos exércitos, aumentando o poderio bélico dos generais. O objetivo do jogo consiste em derrotar todos os demais generais romanos oponentes, destruindo todos os seus exércitos e fortificações.

O mapa é composto por uma espécie de tabuleiro fixo. Cada "espaço" do tabuleiro pode representar 23 tipos diferentes de terreno, que colocados lado a lado formam a figura do mapa. Todas as unidades militares romanas são representadas

por “peças” que se movimentam, ocupando um “espaço” do mapa quando possível. Existem alguns tipos de terreno que são permitidos serem ocupados por uma unidade militar ou fortificação. Tais tipos de terreno representam obstáculos no mapa, como por exemplo árvores, pequenos lagos, óasis, cadeias de montanhas e mares. A Figura 3.1 apresenta todos os tipos de terreno do jogo, que combinados lado a lado formam a figura de um trecho de mapa do jogo.

Figura 3.1: Tipos de terreno do jogo *Alea Jacta Est*

As unidades militares romanas representam agrupamentos de tropas ou exércitos. Cada unidade militar em jogo pode movimentar-se para apenas um “espaço” adjascente por rodada quando esse “espaço” representar um tipo de terreno que permite movimento. Supondo que uma unidade militar possua em sua vizinhança “espaços” que permitem movimento, então existem oito possibilidades diferentes de movimento, situação essa ilustrada na Figura 3.2 (a). Caso existam “espaços”

adjacentes representando obstáculos, como ilustrado na Figura 3.2 (b), então as possibilidades de movimento dessa unidade militar são reduzidas.

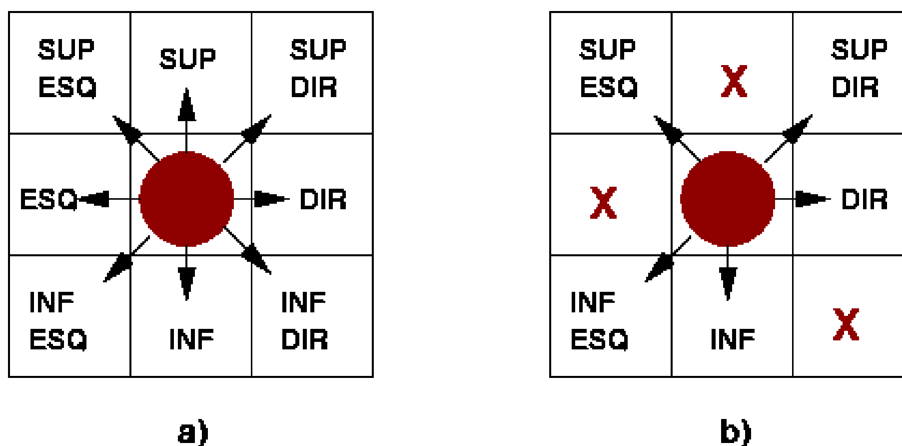

Figura 3.2: Todos os movimentos possíveis para uma unidade militar. a) todos os "espaços" adjacentes permitem movimento. b) apenas alguns "espaços" adjacentes permitem movimento

Cada general pode ter tantas fortificações ou unidades militares quanto conseguir manter sem serem destruídas. Existem 11 tipos diferentes de unidades militares, que representam diferentes tipos de tropas romanas. Cada uma dessas unidades possui três valores numéricos associados, cujo valor é dado de acordo com o tipo da unidade. Esses valores representam respectivamente os atributos **ataque**, **defesa** e **custo de criação**.

O atributo **ataque** representa a força da unidade militar em questão. Esse valor é utilizado para definir a probabilidade de sucesso para vencer um confronto com uma unidade militar inimiga. O atributo **defesa** é utilizado por um elemento de interação atacado para diminuir a probabilidade de sucesso da unidade que está realizando o ataque. Por fim, o atributo **custo de criação** determina quantas rodadas serão necessárias para criar a unidade militar referida. Detalhes sobre o funcionamento de um combate, assim como o modelo probabilístico utilizado para definir o vencedor de um confronto serão dados no capítulo 5.

A Figura 3.3 ilustra todas as unidades militares presentes no jogo *Alea Jacta Est*, assim como o rótulo de cada uma delas e uma tripla ordenada que representa os atributos **(ataque/defesa/custo)** dessa unidade.

A princípio cada jogador inicia a sessão de jogo com apenas uma unidade militar, rotulada *construtor*. A unidade construtor é uma das mais fracas unidades

Figura 3.3: Unidades militares do jogo *Alea Jacta Est*, com seus rótulos e atributos

militares do jogo. No entanto essa unidade possui uma função vital para a manutenção do império do general, pois é através dela que são criadas fortificações. As fortificações devem ser criadas em posições estratégicas do mapa - como por exemplo, próximo de uma cadeia de montanhas de forma a dificultar o ataque dos oponentes - e devem ser protegidas pois são a chave para a expansão do império.

Cada fortificação criada é responsável pela criação de novas unidade militares. A primeira fortificação é criada através da unidade *construtor* que cada general recebe quando participa de uma sessão de jogo. Essa unidade é criada em uma posição aleatória do jogo de forma que cada general inicia o jogo em lugares diferentes do mapa.

Ao escolher um bom local para construir uma fortificação, essa unidade é "sacrificada" e no seu lugar surge uma fortificação. Através dessa fortificação novas unidades militares podem ser criadas, incluindo outras unidades *construtores*. Desses novos construtores surgem novas fortificações, e assim o jogador consegue expandir-se para outras regiões do mapa. A seqüência descrita acima é ilustrada na Figura 3.4.

Diferente das unidades militares, uma fortificação criada tem posição fixa e permite apenas dois tipos de ação: criar novas unidades militares ou abster-se de realizar uma ação. As unidades militares podem movimentar-se para um "espaço" adjascente ou abster-se de seu movimento. A única excessão é a unidade *construtor*, que como comentado acima é capaz de criar novas fortificações no local onde está posicionada.

Quando uma fortificação resolve criar uma nova unidade militar, ela torna-

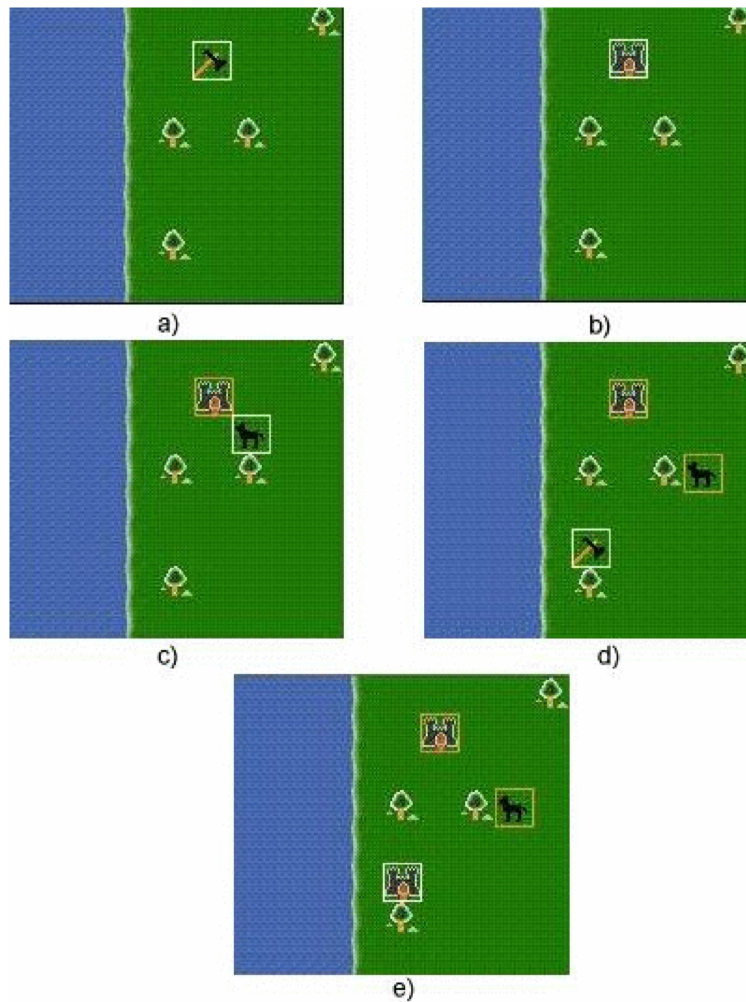

Figura 3.4: Seqüência de criação de novas fortificações. a) Escolha de um local adequado para a fundação. b) Criação da fortificação. c) Criação de uma unidade militar *cavalaria* a partir da fortificação. d) Criação de uma unidade militar *construtor*. d) a unidade *construtor* recém criada constrói uma nova fortificação.

se inativa por uma certa quantidade de turnos. Tornar-se inativa significa ficar indisponível para criar novas unidades ou realizar ações. A quantidade de turnos que a fortificação fica inativa depende do custo da unidade militar que está sendo criada.

Para melhor ilustrar essa situação considere o exemplo onde o general do turno deseja criar um unidade *centurião* em uma de suas fortificações. Após iniciar a construção dessa unidade militar a fortificação ficará indisponível por 3 rodadas a contar de sua próxima rodada de jogo. Caso a unidade a ser criada fosse uma unidade *catapulta*, então a fortificação ficaria inativa por 6 rodadas.

A interação do jogo é feita por rodadas ou turnos. Cada general tem o direito de realizar uma ação para cada unidade militar ou fortificação que lhe pertençam em jogo. As ações que as unidades militares e fortificações podem realizar durante seu turno foram citadas acima. Após cada unidade militar ou fortificação realizar sua ação, a rodada ou turno passa para o próximo jogador/general, e o jogo segue nessa mesma ordem.

Para melhor ilustrar essa interação será apresentado um exemplo onde dois jogadores participam da mesma sessão de jogo. O primeiro jogador possui 3 unidades militares e 2 fortificações em jogo. O segundo jogador possui 7 unidades militares e 5 fortificações em jogo, sendo que dessas 5 fortificações 3 estão inativas - em outras palavras, estão criando novas unidades militares. Quando em seu turno, o primeiro jogador tem o direito de realizar 5 ações ao todo, uma para cada unidade e fortificação sua em jogo. Após realizar as 5 ações, o turno passa para o segundo jogador, que tem o direito de realizar 9 ações apenas - as cidades em construção não participam da rodada.

O confronto entre unidades militares com outras unidades e fortificações inimigas decide o rumo do jogo. É através dos confrontos ou combates que os jogadores oponentes são eliminados do jogo ao terem todas as suas unidade militares e foritificações destruídas.

Um combate é caracterizado quando uma unidade militar de um general tenta ocupar a mesma posição ou "espaço" do mapa que uma unidade militar ou fortificação de um general oponente. Quando isso ocorre inicia-se o confronto, de onde apenas um dos elementos envolvidos sobreviverá. A decisão de quem vence o confronto é dada pelos atributos **ataque** e **defesa** dos elementos envolvidos. No capítulo 5 será apresentado um modelo probabilístico simples que define o vencedor de um confronto. Resumidamente, esse modelo testa a probabilidade de sucesso do atacante com base em seu atributo **ataque**, probabilidade esta que é reduzida pelo atributo **defesa** do defensor. As fortificações nunca atacam, apenas se defendem. Todas as fortificações possuem como valor de atributo **defesa** o nível 2. O elemento derrotado é destruído, minando as forças do general a quem pertencia. A Figura 3.5 ilustra um exemplo de combate.

Vence o jogo o general que conseguir destruir todas as unidades militares e

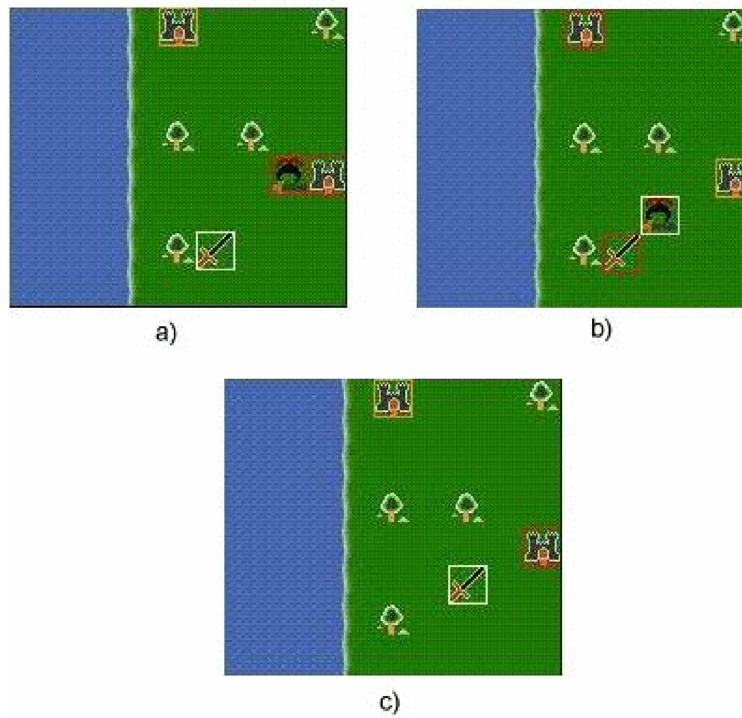

Figura 3.5: Seqüência de jogo envolvendo combate. a) Uma unidade *cohort* aproxima-se de uma unidade *centurião*. b) A unidade *centurião* desloca-se para uma posição adjacente à unidade *cohort*. c) A unidade *cohort* ataca a unidade *centurião*, vencendo o confronto e permanecendo em jogo.

fortificações de todos os generais oponentes que participam da sessão de jogo.

Capítulo 4

Modelagem da Aplicação

Nesse capítulo serão apresentados os aspectos técnicos do jogo *Alea Jacta Est*. Tais aspectos consistem no modelo da aplicação - a interface do jogo, a arquitetura de comunicação utilizada, o formato das mensagens trocada, e o esquema de sincronização dos turnos de jogo. Informações detalhadas sobre a implementação de cada um desses assuntos serão dadas no capítulo 5.

4.1 Interface

Essa seção apresentará a interface para interação entre o jogador e o protótipo. Essa interação inicia-se no momento em que o jogador carrega a aplicação em seu dispositivo móvel, e permanece até o momento em que o jogador decide não mais utilizar a aplicação. Para compreender como ocorre a interação serão descritas abaixo algumas situações que envolvem ações comumente executadas pelo jogador, ilustrando cada uma delas.

4.1.1 Iniciando a aplicação

Para iniciar a aplicação esta deve estar armazenada previamente em um dispositivo móvel. A especificação do perfil MIDP [20] da edição J2ME obriga o fabricante do dispositivo que deseja fornecer suporte a Java a implementar pelo menos uma forma de instalar aplicações desenvolvidas em J2ME por terceiros. Mais do que isso, existe uma forte recomendação para que essa instalação ocorra *on the air* (*OTA*) [21], que facilita a distribuição da aplicação pela Web via WAP.

Partindo do princípio que a aplicação já esteja devidamente instalada no aparelho, este deve fornecer uma forma de carregá-la. Ao ser carregada, a aplicação inicia sua execução, oferecendo ao usuário três opções: iniciar um novo jogo, recorrer a ajuda para ter uma breve descrição das regras e funcionamento do jogo ou sair do mesmo. A interface apresentada ao usuário pode ter uma aparência diferente para cada dispositivo, uma vez que diferentes fabricantes de dispositivo implementam os elementos de interação - também chamados de *widgets* - com a aparência que desejarem. A Figura 4.1 ilustra a interface para o dispositivo colorido padrão disponibilizado pelo emulador *Wireless Toolkit* 2.0 [1].

Figura 4.1: Procedimento de início do jogo. a) tela inicial do jogo. b) menu com as opções iniciais

4.1.2 Criando uma nova sessão de jogo

Quando o usuário decide iniciar um novo jogo, duas possibilidades são apresentadas a ele. A primeira consiste em criar uma nova sessão de jogo enquanto que a segunda permite participar de uma sessão de jogo pré-existente.

Uma observação deve ser feita em relação às sessões de jogo do protótipo implementado. Cada sessão de jogo criada fica em aberto até que seja iniciada pelo jogador que a criou. Durante esse tempo quaisquer jogadores podem ser adicionados na sessão criada para participar do jogo, aguardando o início da mesma. A partir do momento que uma sessão de jogo é iniciada nenhum jogador externo poderá participar dela, apenas podendo criar novas sessões de jogo ou participar de algum sessão ainda em aberto.

[1] `http://java.sun.com/products/j2mewtoolkit`

Para criar uma nova sessão algumas informações são requisitadas ao usuário. Este informa o que é pedido através da digitação em caixas de texto apresentadas no *display* do dispositivo. A entrada de dados pode ocorrer de variadas formas, uma vez que o fabricante do dispositivo é livre para decidir como implementar o tipo de entrada. No entanto, as formas comumente encontradas nesses dispositivos são teclado `QWERTY`, `ITU-T keypad` e *touch screen*, e todas possuem um comportamento padrão independente do dispositivo adotado - que garante a portabilidade da aplicação.

As informações requeridas são o *login* do jogador, o número IP do *host* que está executando a porção servidor do jogo (ver seção 4.2) e o número da porta do serviço nesse *host*. Além disso, o jogador escolhe que deseja criar uma nova sessão de jogo através de uma lista exclusiva. A Figura 4.2 ilustra o procedimento de criação de uma nova sessão de jogo.

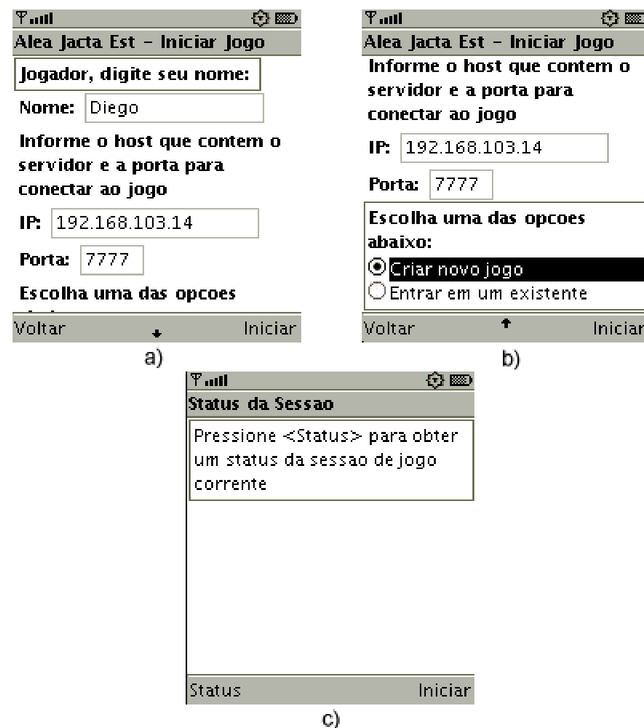

Figura 4.2: Procedimento de criação de uma nova sessão do jogo. a) preenchimento do formulário. b) escolha da opção de criar um novo jogo. c) tela de status e início da nova sessão.

Caso o jogador decida criar uma nova sessão de jogo, a aplicação faz uma

requisição para o servidor criar a sessão. Se for possível criar a sessão, o servidor enviar uma mensagem para a aplicação, informando sobre a criação da mesma.

Após a sessão ter sido criada, o jogador que a criou pode ver o *status* corrente da sessão - ou seja, quantos e quais jogadores pretendem participar da mesma - ou iniciar a sessão de jogo, colocando o jogo em execução.

4.1.3 Entrando em uma sessão de jogo pré-existente

Ao invés de criar uma nova sessão de jogo e aguardar até que a sessão tenha um número satisfatório de jogadores que pretendem participar da mesma, o jogador poderá escolher entrar em uma sessão já existente. Para isso, ele insere as mesmas informações necessárias para o procedimento de criação de uma sessão de jogo. Porém, a opção escolhida da lista será a de entrar em uma sessão pré-existente.

Caso isso ocorra, a aplicação enviar uma requisição para o servidor informar quais são as sessões de jogo em aberto. Ao receber a requisição o servidor envia uma lista com todas as sessões de jogo em aberto para a aplicação. Essa lista contém o *login* do jogador que criou a sessão e a quantidade de jogadores que pretendem participar da mesma. O jogador escolhe qual sessão entrar selecionando uma das sessões da lista, e escolhendo entrar na mesma, ou então decide abortar a operação e retornar para a tela anterior.

Se o jogador entrar em uma sessão de jogo criada, então a aplicação aguarda até que a sessão seja iniciada, colocando o jogo em execução. A Figura 4.3 ilustra o procedimento de participar de uma sessão de jogo em aberto.

4.1.4 Interação durante o jogo

A execução do jogo começa no momento que a sessão de jogo é iniciada. Quando isso ocorre, o servidor avisa a cada um dos jogadores se estes devem realizar alguma ação ou ficar em modo de "espera", aguardando sua vez que jogar. Para ilustrar as possibilidades de interação descritas abaixo será considerado que o jogador está em seu turno e deve realizar ações.

Existem dois tipos de elementos de jogo que permitem interação com o usuário: as unidades militares e as fortificações que a ele pertençam. A forma de interagir com cada um desses elementos é diferente, porém essa interação ocorre através de entradas "padrão" do dispositivo para garantir a portabilidade. Todas as ações são realizadas pelo teclado numérico, seja ele `ITU-T`, `QWERTY` ou *touch screen*. A Figura 4.4 apresenta o teclado `ITU-T` do dispositivo colorido padrão

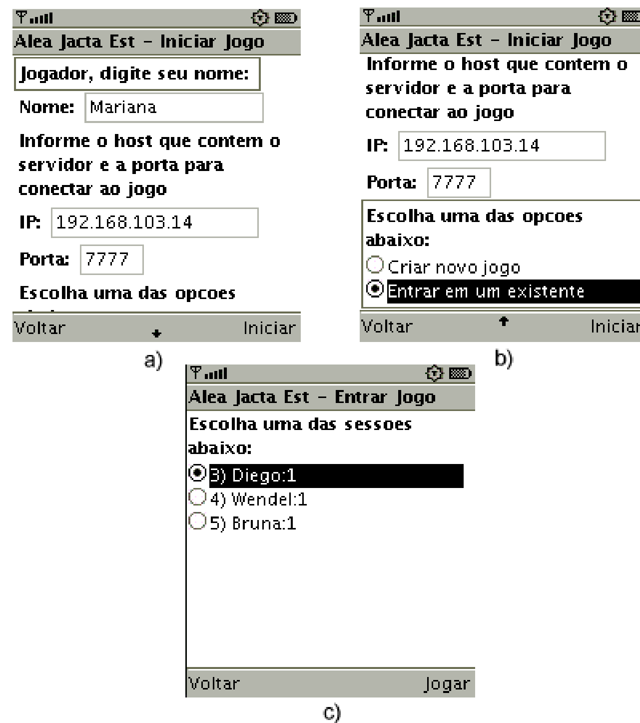

Figura 4.3: Procedimento para participar de uma sessão de jogo pré-existente. a) preenchimento do formulário. b) escolha da opção de entrar em um jogo já criado. c) seleção de uma sessão de jogo em aberto.

do emulador *Wireless Toolkit* 2 utilizado para exemplificar a interação no texto abaixo.

Um observação deve ser feita em relação aos elementos de interação em jogo. O resultado da interação do jogador com a aplicação surte efeito apenas no elemento ativo. O elemento ativo compreende a unidade militar ou fortificação que deve realizar uma ação de jogo no turno corrente. Tal elemento apresenta-se destacado em relação aos demais elementos visíveis na janela de visualização do mapa. Os elementos que pertencem ao jogador aparecem com uma borda de cor laranja, enquanto que os elementos que pertencem aos oponentes sempre aparecem com borda de cor vermelha, independente de quem os possua. O elemento ativo aparece destacado dos elementos do jogador do turno corrente com uma borda de cor branca para indicar que é ele o elemento que deve realizar alguma ação nesse turno.

Figura 4.4: Dispositivo colorido padrão do *Wireless Toolkit* 2.0 com teclado `ITU-T keypad`

As unidades militares podem realizar basicamente duas ações em seu turno: movimentar-se para uma posição adjacente válida ou abster-se de realizar uma ação. Caso o jogador resolva optar por realizar um movimento, ele deve indicar para qual posição adjacente a unidade militar ativa pretende movimentar-se. Decidindo a posição este deve pressionar a tecla numérica correspondente a posição. O protótipo aceita oito teclas numéricas para movimentar uma unidade militar. Tais teclas são as correspondentes aos números `4`, `6`, `8`, `2`, `7`, `9`, `1` e `3`, e correpondem respectivamente aos movimentos a esquerda, a direita, acima, abaixo, abaixo e a esquerda, abaixo e a direita, acima e a esquerda, acima e a direita, movimentos estes em relação a posição ocupada pela unidade militar ativa.

Ao selecionar a posição a movimentar-se a aplicação envia ao servidor uma requisição de movimento, indicando qual a posição que pretende ocupar do mapa. O servidor verifica se a posição solicitada é inválida. Caso verdadeiro, informa ao jogador que o movimento não é permitido, passando a vez para a próxima unidade

militar com ação a realizar.

Caso a posição seja válida, o servidor verifica se existe alguma outra unidade militar ou fortificação oponente na posição desejada. Caso exista alguma, então o movimento configura um confronto de onde apenas um dos elementos envolvidos permanece em jogo. Se o ataque for bem sucedido e não existir mais nenhuma unidade ou fortificação oponente na posição requerida então a unidade militar ativa do jogador do turno ocupa a posição solicitada. Caso contrário, esta permanece onde está devido a presença de outras unidades ou fortificação inimigas na posição especificada.

Caso o jogador queira abster-se da ação de alguma unidade militar, este deve fazê-lo pressionando a tecla correspondente ao número `5`. Caso isso ocorra, a aplicação enviará ao servidor uma notificação de que não irá realizar nenhuma ação para a unidade militar ativa no turno corrente, passando a vez para a próxima unidade militar pertencente ao jogador ou fortaleza caso este não possua mais nenhuma unidade com ação a realizar.

Um caso especial ocorre quando a unidade militar ativa é um *construtor*. Como descrito no capítulo 3, a unidade *construtor* permite a construção de fortificações. Caso o jogador decida criar uma fortificação a partir de uma unidade *construtor* ativa, este deve fazê-lo pressionando a tecla `0`. A aplicação envia ao servidor uma requisição de construção de fortificação, informando qual a unidade construtor que deve ser "sacrificada" na construção. Ao identificar tal unidade, o servidor envia uma notificação para a aplicação informando a destruição da unidade *construtor* especificada, seguida de uma notificação informando sobre a criação de uma nova fortificação na posição antes ocupada pelo *construtor* destruído. A figura 4.5 ilustra o procedimento de criação de fortificações a partir de uma unidade *construtor*.

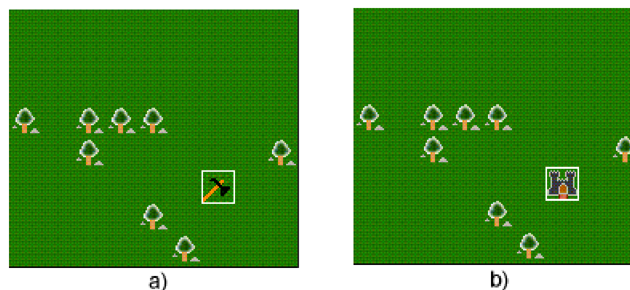

Figura 4.5: Sequência de criação de uma fortificação. a) a unidade *construtor* escolhe uma posição no mapa. b) a unidade *construtor* é "sacrificada" para dar lugar a uma nova fortificação.

Quando o elemento de interação ativo for uma fortificação, apenas duas opções

de ação estão disponíveis. A primeira consiste em abster-se de relizar qualquer ação de jogo, de forma semelhante ao que ocorre com as unidades militares. Tal ação é realizada pressionando-se a tecla correspondente ao número `5`. A seqüência de comunicações executada entre a porção cliente e servidor é equivalente ao caso acima, não sendo necessário ilustrá-la.

A outra ação de jogo possível para uma fortificação consiste em utilizá-la para criar novas unidades militares e aumentar os exércitos do jogador. Isso é realizado através do pressionamento da tecla correspondente à opção `Criar`, que muitas vezes é mapeada como um *soft-button* do aparelho. Ao ser pressionada a porção cliente exibe uma nova tela onde é permitido escolher qual é o tipo de unidade militar a criar. A aplicação então envia uma requisição ao servidor solicitando a criação dessa nova unidade militar a partir da fortificação especificada.

A quantidade de turnos de jogo necessários para criar uma unidade militar varia de unidade para unidade, como visto no capítulo 3. O servidor enviará uma notificação nula para a aplicação a cada fim de turno para cada fortificação cujo estado seja *em construção*. Após percorridos os turnos necessários a aplicação apresenta uma nova unidade militar, que fica disponível para realizar ações quando estiver ativa. A Figura 4.6 ilustra o procedimento de criação de novas unidades a partir de uma fortificação.

4.2 Arquitetura de Comunicação

É comum encontrar na literatura sobre desenvolvimento de jogos *multiplayer* para computadores *desktop* dois tipos de arquitetura de comunicação extensivamente utilizadas. Tratam-se das arquiteturas de comunicação ponto-a-ponto (a) e cliente-servidor (b) ilustradas na Figura 4.7. Cada uma dessas arquiteturas possui vantagens e desvantagens que variam de acordo com as necessidades da aplicação a ser desenvolvida.

Segundo [22] a arquitetura ponto-a-ponto é muito utilizada para jogos que empregam uma visão *top-down*, como é o caso de jogos de estratégia de tempo-real. Para jogos que implementam essa arquitetura de comunicação é tolerável uma certa quantidade de *delay* desde que uma grande quantidade de unidades sejam sincronizadas.

Nessa abordagem cada participante executa uma simulação do jogo em sua própria máquina. A largura de banda é reduzida transmitindo-se eventos de entrada para os pontos da arquitetura ao invés das unidades propriamente ditas. Além disso cada jogador deve manter uma conexão com os demais jogadores para troca de

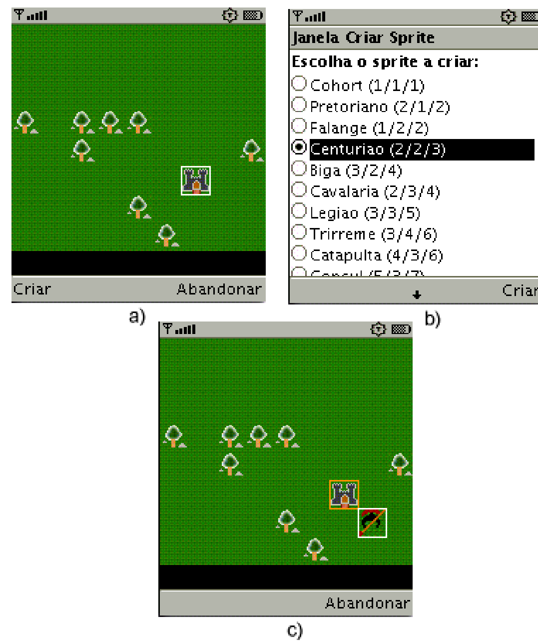

Figura 4.6: Sequência de criação de novas unidades militares a partir de uma fortificação. a) o jogador decide criar uma unidade a partir da fortificação ativa. b) o jogador seleciona a unidade militar *centurião* de uma lista de unidades militares possíveis. c) Após percorridos 3 turnos de jogo a unidade *centurião* é criada e disponibilizada para o jogador.

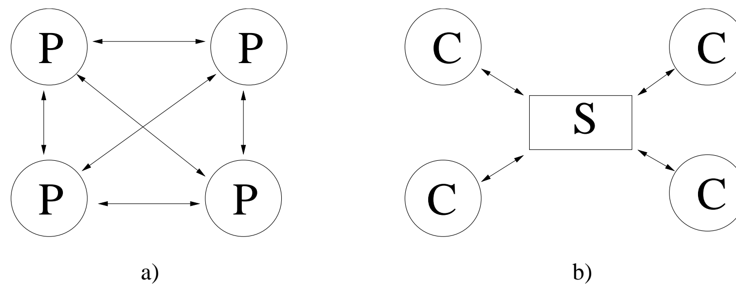

Figura 4.7: Arquiteturas de comunicação clássicas utilizadas em jogos *multiplayer* para computadores *desktop*. a) arquitetura ponto-a-ponto. b) arquitetura cliente-servidor.

mensagens.

Na arquitetura cliente-servidor os clientes coletam eventos de entrada e realizam a renderização da cena com base no estado do jogo fornecido pelo servidor.

Toda a carga de processamento é deixada para a porção servidor, que envia o resultado de uma ação que altera o estado do jogo para os clientes. Além disso, nesse tipo de arquitetura os jogadores não trocam informações diretamente uns com os outros, mas o fazem através do servidor.

Embora jogos de estratégias que empregam visão *top-down* utilizem massivamente a arquitetura ponto-a-ponto, esse protótipo foi implementado utilizando-se a arquitetura cliente-servidor devido a vários motivos que a princípio tornam tal arquitetura mais adequada para implementação de jogos *multiplayer* e outras aplicações distribuídas em dispositivos móveis. Dentre esses motivos os principais consistem em:

- Tal arquitetura em geral apresenta-se mais escalável que a ponto-a-ponto quando o número de usuários cresce [23] [24] permitindo que muitos jogadores participem de uma única sessão do jogo ao mesmo tempo sem criar uma perda de desempenho do jogo;

- Os aparelhos MID (*Mobile Information Devices*) em geral apresentam severas restrições em aspectos como poder de processamento e memória, que impedem que aplicações desenvolvidas em J2ME sejam muito robustas. Acrescentar processamento adicional a tais aparelhos pode comprometer certos aspectos do jogo, como por exemplo a interatividade, tornando-o pouco atrativo.

  Além disso as restrições de memória do MID impedem que grandes quantidades de informações sejam armazenadas nesses aparelhos, de forma a ser inviável manter a consistência do jogo em cada aparelho como exigido pela arquitetura ponto-a-ponto. Segundo a especificação MIDP 1.0 [20], cada MID deve possuir no mínimo 128K de memória não volátil (*flash card* ou ROM) para o software MIDP, 8K de memória não volátil para armazenamento definido pela aplicação e 32K de memória volátil (RAM) para a pilha de execução do Java. Tais quantidades são adicionais a quantidade de memória requerida pela configuração CLDC [25] e restringem a quantidade de dados que podem ser armazenados em uma aplicação móvel, tornando a arquitetura cliente-servidor mais adequada para jogos *multiplayer* executando em aparelhos móveis. Dessa forma, grande parte do processamento será executado na porção servidor executando em uma máquina *desktop* conectada à Internet.

- Embora a edição J2ME seja adequada ao desenvolvimento de aplicações *smart-client*, permitindo que muito do processamento seja realizado pela

porção cliente, utilizar a arquitetura de comunicação cliente-servidor para jogos multiusuário divide melhor as tarefas de cada porção da arquitetura. Dessa forma é permitido que a aplicação cliente apenas preocupe-se com a renderização de imagens no *display* do MID e transmissão das ações de cada jogador, enquanto que a porção servidor preocupar-se-á em manter a consistência do jogo e notificar os demais clientes sobre o novo estado do jogo após o cômputo do resultado de alguma ação realizada pelo jogador do turno.

Além disso existem diversas outras vantagens em adotar essa arquiterura para certas categorias de jogos interativos *multiplayers*. Abaixo enumeraremos algumas dessas vantagens extraídas de [22] e [26]:

- Permite que muito da computação do estado do jogo seja realizada pelo servidor, de forma a garantir facilmente a consistência global do jogo;

- Cada cliente deve manter apenas uma única conexão com o servidor, independente de quantos jogadores participam do jogo corrente, que permite diminuir o tráfego na rede uma vez que cada jogador envia a mensagem apenas uma vez para o servidor. Além disso algumas informações são relevantes apenas para alguns jogadores, não sendo necessário enviá-las a todos os jogadores que participam de uma sessão de jogo;

- Possibilidade do servidor comprimir os mensagens antes de enviá-la de forma a minimizar o tráfego da rede;

- Capacidade de prevenir *cheating* e *hacking*. A técnica de *cheating* consiste na exploração de *bugs* do jogo, permitindo ao jogador obter injustamente uma vantagem sobre os demais usando intensionalmente a falha a seu favor, aproveitando-se das fraquezas da arquitetura do jogo[2].

  Hacking consiste em uma técnica de trapaça semelhante ao *cheating*, porém essa ocorre através da alteração de um código que já tenha sido compilado [3].

  As duas técnicas, comuns em jogos ponto-a-ponto, são prevenidas na arquitetura cliente-servidor pois os jogadores não podem manipular cálculos envolvidos em decisões uma vez que tais cálculos são realizados em um servidor para o qual os jogadores não tem acesso;

[2] `http://www.ke9.com/guides/ooc/cheating.html&e=747`
[3] `http://www.accessorl.net/~cyberwar/codehacks.html#hacking`

- O jogo pode ser alterado ou estendido pelo desenvolvedor sem a necessidade de atualizar o *software* cliente na máquina do jogador. Tais atividades incluem fixação de erros e adição de novos níveis ou mapas ao jogo.

- O uso de servidor introduz um sincronização natural entre os jogadores.

Embora essa arquitetura possua diversas vantagens, algumas grandes desvantagens surgem dessa comunicação entre o cliente e o servidor. Abaixo são apresentadas algumas que melhor exploram as fraquezas dessa arquitetura, extraídos de [27]:

- Em uma arquitetura centralizada cliente-servidor a informação alcança seu destino apenas através do servidor, que aumenta a latência da rede;

- Todos os dados convergem para um servidor centralizado, tornando-o um "gargalo". Uma vez que o servidor deve coletar os dados de todos os jogadores, computar o estado do jogo e servir a todos os participantes do jogo, a freqüência na qual o estado do jogo é computado diminui com o uso intenso da CPU podendo causar uma perda na interatividade do jogo. Esse problema é agravado quando o número de jogadores aumenta.

  Tal problema pode ser minimizado com a adoção de múltiplos servidores e uso de balanceamento de carga entre eles, porém o problema ainda existe devido a limitações da potência de processamento desses servidores e da largura de banda da rede que os conectam.

- Dependendo do mecanismo de sincronização utilizado em uma aplicação desenvolvida para essa arquitetura, uma falha em um dos clientes participantes pode causar grandes problemas para o desenrolar do jogo.

Devido a tais problemas, não é uma boa escolha implementar a arquitetura cliente-servidor para algumas categorias de jogos interativos. Preferencialmente os jogos de interação em primeira pessoa são implementados utilizando essa abordagem devido a pequena quantidade de elementos que necessitam ser sincronizados e da baixa latência requerida por tais jogos. Exemplos de jogos que utilizam essa arquitetura incluem os clássicos *Quake* e *EverQuest*.

Utilizar essa arquitetura para o desenvolvimento do protótipo consistiu em um trabalho puramente experimental, uma vez que tal arquitetura não é a escolha clássica para essa categoria de jogos. No entanto, devido ao fato do jogo *Alea Jacta*

Est ser utilizado em dispositivos muito restritos, que impede que grandes quantidades de dados sejam armazenadas em cada MID, e sincronizado por turnos, não exigindo o mesmo tempo de resposta que jogos interativos de tempo-real essa escolha justifica-se.

Na implementação do protótipo utilizou-se apenas um servidor para gerenciar as sessões de jogo em execução e as conexões oriundas de novos jogadores. Tal implementação foi suficiente para ilustrar o propósito desse trabalho. Quaisquer mensagens trocadas entre dois jogadores devem ser enviadas ao servidor do jogo, que processa-as e envia para o jogador de destino, centralizando o tráfego de mensagens.

Para conectar ao jogo o jogador deve informar o número IP do *host* que irá executar a porção servidor do jogo. O número da porta do serviço utilizado no protótipo foi escolhido como 7777. A Figura 4.8 ilustra a arquitetura do protótipo do jogo *Alea Jacta Est*.

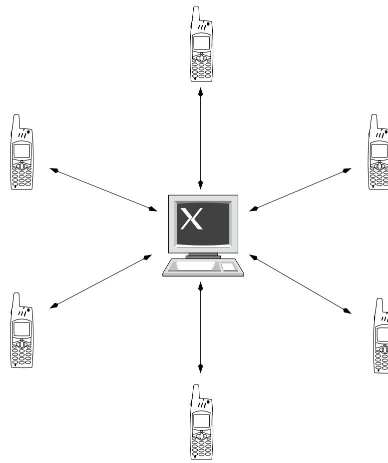

Figura 4.8: Arquitetura cliente-servidor utilizada no protótipo *Alea Jacta Est*

A porção cliente do jogo possui apenas a preocupação de "escutar" ações do jogador, comunicando-as ao servidor, além de renderizar os resultados das ações no *display* do dispositivo. A porção servidor do jogo realiza tarefas mais complexas e que requerem um poder computacional maior, sendo um dos principais motivos para a adoção dessa arquitetura.

Dentre as atividades realizadas pelo servidor, destacam-se:

- Gerenciar as sessões de jogos em execução. Mais que uma sessão de jogo

pode estar em execução, e é tarefa do servidor gerenciá-las. Para cada jogo em execução existe uma `thread` diferente executando, de forma que o desenrolar de cada sessão de jogo é totalmente independente das demais. Isso também permite que novas sessões do jogo sejam abertas a qualquer momento por qualquer cliente conectado ao servidor que não esteja em jogo;

- Gerenciar as rodadas do jogo, sincronizando as ações dos jogadores. Apenas um jogador realiza ações por turno. Um turno caracteriza a realização de uma ação para cada unidade militar e fortificação cujo estado seja *fora de produção* de um jogador. Somente após a execução de todas as ações pertencentes ao jogador do turno corrente o servidor passa a vez para o próximo jogador da sessão, até que todos os jogadores que dela participam tenham efetuado suas ações, caracterizando uma rodada de jogo.

- Processar e validar as ações de um dado jogador, que inclui verificar se o movimento de uma unidade militar desse jogador é válido, se um ataque realizado por esse jogador foi bem sucedido ou não, ou se uma fortificação desse jogador ainda está em produção. Para cada ação processada existe uma resposta na forma de notificação para o cliente que solicitou-a. Os demais jogadores que dependem do resultado das ações em questão serão informados sobre as modificações no estado do jogo quando estiverem em seu turno.

  Dessa forma cada aplicação cliente mantêm consistência local, conhecendo apenas as características dos trechos de mapa onde existe algum elemento de interação que lhe pertença posicionado. A consistência global do jogo é conhecida apenas pelo servidor, que centraliza todo o processamento e lógica do jogo.

4.3 Formato de Mensagens

Uma vez que a arquitetura de comunicação escolhida para a implementação do protótipo *Alea Jacta Est* foi a arquitetura cliente-servidor, foi necessário criar um conjunto de "primitivas" de comunicação para serem trocadas entre a porção cliente e a porção servidor. Todas as ações realizadas pelo jogador do turno corrente são convertidas nessas primitivas e enviadas na forma de mensagens ao servidor para serem então processadas.

Para este protótipo foram projetadas diversas primitivas de comunicação. Tais primitivas formam o protocolo de comunicação do jogo, e é através delas que a

porção servidor sabe quais alterações devem ser feitas no estado do jogo.

Cada mensagem possui um formato bem definido que é dado pela Figura 4.9. Cada mensagem é composta de uma *string* de caracteres divididos em dois campos: o cabeçalho da mensagem e o corpo da mensagem. O cabeçalho consiste em uma seqüência de três caracteres que representa a quantidade de caracteres presentes no corpo da mensagem, variando de 0 a 999. O corpo da mensagem contém uma primitiva de comunicação com seus parâmetros, todos separados por um caracter ponto-e-vírgula (;). Quando uma mensagem é recebida por uma das porções da arquitetura esta é decodificada de acordo com a primitiva enviada. Toda a comunicação entre os clientes e o servidor ocorre através da troca de mensagens que codificam ações ou notificações.

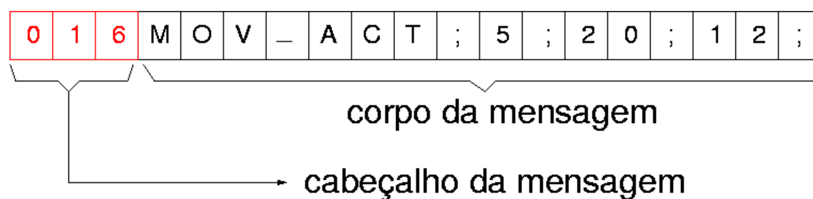

Figura 4.9: Formato das mensagens trocadas entre a porção cliente e servidor do protótipo

As primitivas envolvidas na comunicação das porções do jogo são classificadas em três categorias: primitivas de ação, primitivas de notificação e primitivas de gerência de sessão de jogo.

As primitivas de gerência de sessão são utilizadas para a aplicação cliente solicitar ao servidor a criação de novas sessões de jogo, entrar em uma sessão pré-existente, obter o *status* corrente da sessão em aberto e iniciar uma sessão de jogo. Abaixo serão brevemente descritas cada uma dessas primitivas.

- `NOV_ACT`: utilizada para solicitar ao servidor a criação de uma nova sessão de jogo.
- `SES_NOT`: primitiva enviada como resposta a uma solicitação de criação de sessão de jogo ou em resposta a solicitação de participar de uma sessão de jogo pré-existente.
- `SES_ACT`: primitiva utilizada para solicitar ao servidor a participação de uma sessão de jogo pré-existente.
- `LIS_ACT`: primitiva enviada para o servidor solicitando uma listagem de todas as sessões de jogo em aberto.

- `LIS_NOT`: primitiva enviada como resposta a uma solicitação `LIS_ACT`. A listagem enviada contém o número da sessão em aberto, o *login* do jogador que criou a sessão e o número de jogadores aguardando o início da mesma.

- `EST_ACT`: primitiva enviada pelo jogador que criou uma sessão solicitando o *status* da mesma.

- `EST_NOT`: primitiva enviada pelo servidor como resposta a uma solicitação `EST_ACT`. A mensagem com essa primitiva contém uma listagem de todos os jogadores participantes da sessão aberta.

- `COM_ACT`: primitiva enviada pelo jogador que criou a sessão solicitando o início da mesma.

- `COM_NOT`: primitiva enviada pelo servidor para o jogador que realizou uma solicitação `COM_ACT` indicando o início do jogo.

Existem 7 primitivas utilizadas para codificar ações do jogador. Tais primitivas variam em número de parâmetros e significado dos mesmos. Abaixo será descrito cada uma delas, sendo que uma explicação detalhada sobre todas as primitivas será dado no capítulo 5:

- `MOV_ACT`: primitiva utilizada para codificar uma ação envolvendo o movimento de alguma unidade militar. Os parâmetros informados pelo cliente para essa primitiva são qual é a unidade militar que deseja efetuar um movimento e qual é a posição que essa unidade pretende ocupar do mapa.

- `CID_ACT`: primitiva utilizada na codificação de uma requisição enviada ao servidor solicitando a construção de uma nova fortificação a partir de uma unidade *construtor*. A única informação enviada com essa primitiva consiste em qual é a unidade *construtor* que será "sacrificada".

- `SPR_ACT`: primitiva utilizada para codificar uma requisição para construir unidades militares a partir de uma fortificação pré-existente. Os parâmetros informados com essa primitiva consiste em qual é a fortificação que deve construir a unidade militar e qual é a unidade a ser construída.

- `MAP_ACT`: primitiva utilizada para codificar uma requisição para o servidor enviar um trecho do mapa do jogo. O parâmetro enviado com essa primitiva qual trecho do mapa a enviar.

- INI_ACT: utilizada para solicitar ao servidor uma listagem de unidades militares oponentes pertencentes a um mesmo trecho de mapa.
- CIN_ACT: primitiva utilizada para solicitar ao servidor uma lista com todas as cidades oponentes encontradas em um dado trecho de mapa.
- NUL_ACT: utilizada para informar que o elemento de interação ativo não vai realizar nenhuma ação nesse turno. Não necessita de nenhum parâmetro adicional.

Quando o servidor recebe uma mensagem oriunda de um cliente esta é decodificada. A partir de sua primitiva, extrai-se os parâmetros da mensagem e realiza-se a ação no estado corrente do jogo mantido pelo servidor. Cada ação do cliente gera uma notificação do servidor.

Uma notificação é uma mensagem que informa ao jogador que a recebe alguma mudança ocorrida no estado do jogo. Existem várias primitivas que codificam uma notificação do servidor, que serão brevemente descrita abaixo:

- MOV_NOT: primitiva utilizada para indicar a realização de um movimento. A mensagem enviada com essa primitiva guarda a nova posição da unidade militar que requisitou o movimento.
- SPR_DES: primitiva utilizada para indicar ao jogador que uma unidade militar que lhe pertence foi destruída e está fora de jogo. Essa situação pode ocorrer quando uma unidade militar é atacada e perde o confronto ou quando uma unidade *constutor* é "sacrificada" para construir uma nova fortificação.
- MAP_NOT: primitiva utilizada para informar a um jogador quais são os tipos de terreno que pertencem ao trecho de mapa solicitado.
- SPR_NOT: primitiva utilizada para informar que uma unidade militar foi criada a partir de uma fortificação. O tipo de unidade e qual fortificação criou-a são informados no corpo da mensagem.
- INI_NOT: primitiva enviada a um cliente contendo uma lista com todas as unidades militares inimigas presentes em um dado trecho de mapa.
- CIN_NOT: primitiva enviada a um cliente contendo uma listagem de cidades inimigas pertencentes a um trecho de mapa solicitado.
- CID_DES: primitiva enviada a um jogador informando que uma de suas fortificações foi destruída.

- ERR_NOT: primitiva enviada a uma aplicação cliente em resposta a uma ação impossível de ser realizada, como por exemplo tentar criar uma fortificação através de uma unidade militar que não seja um *construtor*.

- NUL_NOT: primitiva enviada em resposta a uma ação nula. Também é utilizada para informar a um jogador no final de seu turno que uma de suas fortificações ainda permanece no estado *em contrução*.

- FIM_NOT: primitiva utilizada para sinalizar o fim do jogo.

- WAI_NOT: primitiva enviada a um jogador para indicar que este deve entrar em estado de "dormência", apenas ouvindo por notificações do servidor.

- TUR_NOT: primitiva enviada para um jogador para sinalizar o início de seu turno de jogo.

Uma observação a fazer é que para cada ação que um cliente envia em seu turno para o servidor ele recebe uma notificação correspondente. Nessa categoria encontram-se as primitivas MOV_ACT, MAP_ACT, SPR_ACT, CID_ACT, INI_ACT, CIN_ACT e NUL_ACT. Porém, quando a aplicação cliente não está em seu turno - e sim no estado de "dormência" - ela pode receber notificações do servidor sem que uma ação tenha sido solicitada. Nesse estado as notificações possíveis de serem recebidas são SPR_DES, CID_DES, NUL_NOT, FIM_NOT, WAI_NOT e TUR_NOT, todas descritas acima.

A sintaxe completa de cada uma da mensagens descrita nessa seção será apresentada no Apêndice A.

4.4 Sincronização dos Turnos

O protótipo foi desenvolvido para ser um jogo sincronizado por turnos de forma que cada jogador realiza ações que modificam o estado do jogo apenas quando estiverem em seu turno. Quando em seu turno o jogador tem o direito de realizar uma ação para cada unidade militar e fortificação que não esteja no estado *em construção*.

Se um jogador realiza todas as ações permitidas ele encerra seu turno. Quando isso ocorre o servidor envia uma mensagem WAI_NOT para esse jogador, fazendo com que entre no estado de "dormência". A partir desse momento o jogador não mais realiza ações, apenas recebe notificações oriundas do servidor.

Além de colocar o jogador que encerrou o turno no estado de "dormência", o servidor decide quem é o próximo jogador que terá o turno de jogo, enviando uma notificação `TUR_NOT` para o mesmo. Desse momento em diante a aplicação cliente desse jogador estará pronta para receber ações, codificá-las em mensagens e enviá-las ao servidor, esperando receber as notificações de acordo com a ação executada.

A aplicação cliente do jogador do turno realiza um mesmo procedimento para cada ação que o jogador tem direito. As unidades militares pertencentes ao jogador devem realizar suas ações primeiro, seguido de uma ação para cada fortificação no estado *fora de produção*.

O procedimento de renderização é realizado para cada elemento de interação que tornar-se-á ativo durante o turno. Primeiro a aplicação cliente identifica quem é esse elemento. Em seguida é solicitado ao servidor todos os tipos de terreno do mapa que pertencem ao trecho onde o elemento está presente e uma listagem com todas as fortificações e unidade militares inimigas presentes nesse trecho. Com base nessas informações a renderização é feita no *display* do dispositivo móvel, desenhando todos os terrenos do trecho, as fortificações inimigas, as unidades militares inimigas e as fortificações e unidades militares pertencentes ao jogador do turno. Por fim, a unidade de interação ativa é destacada das demais para indicar que a ação realizada surtirá efeito apenas sobre ela.

O turno de jogo é a unidade básica de sincronização adotada no protótipo. Durante um turno todos os jogadores realizam ações e obtêem resultados das ações dos demais jogadores aplicadas no estado do jogo. O turno também é utilizado para computar quanto tempo uma fortificação deve ficar inativa antes de contruir uma unidade militar solicitada. Maiores detalhes sobre a implementação da sincronização por turnos serão apresentados no capítulo 5.

Capítulo 5

Implementação

Este capítulo tem por objetivo apresentar o funcionamento do protótipo, explicando com detalhes como o servidor gerencia a lógica e estado do jogo e como as mensagens contendo as primitivas de comunicação são geradas em resposta a ações do jogador, ilustrando as situações onde isso ocorre. A lógica implementada no protótipo do jogo *Alea Jacta Est* segue estritamente as informações contidas nesse capítulo.

5.1 O Estilo do Jogo

O protótipo *Alea Jacta Est* foi concebido como o protótipo de um jogo *top-down view* onde toda a interação entre o jogador e o sistema ocorre através da realização de ações bem definidas para cada elemento de interação que participa do jogo. Tal protótipo foi baseado em jogos de estratégia clássicos como *Civilization* [1] e *Age of Empires* [2], onde o objetivo do jogo é vencer todos os jogadores adversários derrotando seus impérios.

A visão *top-down* foi escolhida para o protótipo devido a facilidade de interação com o ambiente do jogo que ela oferece ao jogador. Esse tipo de visão permite que o jogador tenha uma visão de todo o ambiente que cerca seus elementos de interação. Dessa forma é permitido aproveitar os obstáculos do terreno contidos no mapa para criar boas estratégias de ataque e defesa.

---

[1] `http://www.infogames.com.br/jogos.asp?jogoid=17`

[2] `http://www.microsoft.com/catalog/display.asp?subid=22&site=10977&x=45&y=11`

Outro grande motivo para a adoção desse tipo de visão e estilo de jogo no protótipo está relacionado às limitações do dispositivo que executará a aplicação. Jogos de interação em primeira pessoa - como os clássicos *Doom* e *Quake* - realizam renderização em objetos 3-D. Tal procedimento necessita de processamento relativamente grande, não sendo adequado para utilização de dispositivos que utilizam processadores de propósito muito específico como aparelhos celulares. Além disso a interação entre o jogador e o jogo é um pouco mais complexa nessa categoria de jogos, tornando inviável sua implementação.

Os objetos perceptíveis na visão *top-down* tecnicamente dividem-se em duas categorias: os *tiles* e os *sprites*. Um *tile* consiste em um objeto estático incapaz de realizar qualquer tipo de interação com o usuário. Para esse protótipo podem ser considerados *tiles* os objetos que representam o tipo de terreno, pois a única função que possuem é ilustrar o ambiente onde a interação ocorre. Jogos de computador baseados em "tabuleiro" são também chamados *tile-based* [28] [29] devido a essa característica onde cada elemento de interação ocupa um único espaço do "mapa", espaço este representado por uma *tile*.

Os *sprites* por sua vez são objetos que permitem algum tipo de interação com o usuário. Muitas vezes são objetos animados do jogo, e podem tanto pertencer a um jogador ou fazerem parte dos oponentes controlados pelo sistema. Para esse protótipo consideraremos como *sprite* as unidades militares em jogo e as fortificações, que apesar de fixas permitem algum tipo de interação - no caso, criar novas unidades militares a escolha do jogador do turno corrente.

Detalhes sobre a manipulação dos *tiles* e *sprites* pela porção servidor serão dados nas seções abaixo.

5.2 O Estado do Jogo

Como comentado anteriormente, a porção servidor do protótipo é a responsável pela manutenção da consistência do estado do jogo. Ela recebe mensagens contendo primitivas de ação do jogador do turno, processa as ações e gera mensagens contendo primitivas de notificação que informam aos jogadores sobre as mudanças ocorridas no estado do jogo.

Para realizar tal processamento a porção servidor utiliza estruturas de dados para armazenar o estado do jogo e efetuar processamento. Tal estado consiste nos elementos que participam de uma sessão de jogo - as unidades militares, as fortificações e os jogadores.

Essencialmente esse armazenamento consiste no próprio estado do jogo. Al-

terações no estado incluem inserir novos jogadores na sessão enquanto ela estiver em aberto, inserir e remover unidades militares e fortificações para cada jogador da sessão quando em jogo e alterar o estado de cada fortificação para *em contrução* ou *fora de construção*.

Para facilitar a manipulação desses dados os objetos foram agrupados em algumas estruturas de dados que serão descritas nas seções 5.2.1, 5.2.2 e 5.2.3, cuja manipulação será minunciosamente descrita na seção 5.3.

5.2.1 O Mapa

O mapa do jogo é utilizado para definir o ambiente onde a interação do jogo ocorre. No contexto do *Alea Jacta Est* este representa o mundo onde os generais romanos podem expandir seus exércitos. Tal mapa possui alguns obstáculos naturais - florestas, oásis, cadeias de montanhas, pequenos e grandes lagos e mares - que podem ser utilizados pelos jogadores para criar boas estratégias de ataque e defesa.

Em nível de implementação um mapa pode ser considerado uma matriz de tipos de terreno. No protótipo *Alea Jacta Est* este mapa é uma matriz de dimensão 100x100. Cada tipo de terreno, como citado acima, pode ser considerado um *tile* pois tal elemento não apresenta interação com o jogador, exceto no caso dos obstáculos que impedem o movimento de uma unidade militar para a posição em que existem.

Armazenar uma matriz de dimensão 100x100 em uma máquina *desktop* em geral não consiste um problema, mas em se tratando de dispositivos móveis essa quantidade de memória pode estar muito além do que é disponibilizado para o sistema. Na seção 5.2.1 será apresentada uma forma de lidar com essa restrição, permitindo que mesmo um dispositivo com quantidades restritas de memória seja capaz de permitir interação com um mapa de grande dimensões.

As seções abaixo apresentam a representação utilizada para dois importantes elementos implementados nesse protótipo: o *tile* e o trecho de mapa. A combinação desses elementos constitui a estrutura de dados que armazena o mapa utilizado pelo protótipo.

Os *Tiles*

Os *tiles* consistem no "bloco de construção" básico de um mapa. Eles representam os tipos de terreno disponíveis para a construção do mapa. Cada *tile* em um jogo *tile-based* é análogo a uma "casa" de um jogo de tabuleiro. Uma unidade de interação ocupa apenas um casa por rodada, sendo que o movimento de cada unidade

ocorre nas casas adjacentes à casa por ela ocupada.

Alguns *tiles* não permitem que um elemento de interação os ocupem. É o caso dos tipos de terreno que representam obstáculos do mapa. Em nível de implementação cada elemento que representa um tile deve possuir algumas pelo menos as seguintes características:

- `idTile`: utilizado para armazenar qual dos tipos de terreno possíveis o *tile* representa. Para o caso desse protótipo esse ID deve possuir um valor entre `0` e `23`, como visto no capítulo 3.
- `flagMovimento`: é aconselhável possuir um *flag* que indica se o *tile* permite ou não movimento de elementos de interação na "casa" do mapa que ele representa.

É interessante associar um *tile* a um elemento gráfico que ilustra o tipo de terreno do mapa por ele representado. Isso pode ser feito na definição do tipo ou em tempo de execução. Não é muito vantajoso associar uma imagem a um *tile* na definição desse tipo de dados quando o mapa do jogo utilizado pela aplicação é muito grande, pois a redundância pode ocupar uma quantidade de armazenamento vital para a aplicação.

Para o protótipo foi implementado no dispositivo móvel uma estratégia análoga ao sistema de *cache*, onde as imagens que representam o tipo de terreno foram armazenadas em memória apenas uma vez e renderizadas quantas vezes necessário. Embora a redundância ainda permaneça na representação do mapa, essa abordagem minimiza a quantidade de memória necessária para o armazenamento do mesmo, uma vez que cada imagem utilizada possui dimensão de 20x20 *pixels* e representação da ordem de algumas centenas de *bytes*.

Os Trechos de Mapa

Os dispositivos cobertos pela configuração CLDC e perfil MIDP da edição J2ME são bastante restritos em termos de armazenamento. Dessa forma é inviável armazenar o mapa completo do jogo na memória de cada dispositivos.

A solução encontrada consiste em segmentar o mapa em trechos menores, cada um com dimensão 20x20. Dessa forma o mapa original será composto por 25 trechos de mapa diferentes, numerados de `0` a `24`. Para cada trecho rotulado existe uma posição (x, y) inicial associada, de forma que tais trechos colocados lado a lado formam o mapa completo com dimensão 100x100. A Figura 5.1 ilustra essa representação.

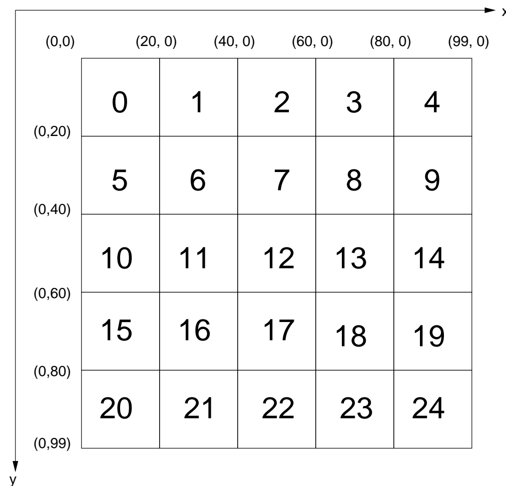

Figura 5.1: Representação do mapa do jogo por trechos

Além de resolver o problema de armazenamento nos dispositivos móveis, utilizar trechos de mapa facilita o controle da interação do jogo. Um exemplo disso ocorre quando a aplicação cliente do jogador do turno deve fazer a renderização de uma cena de jogo. Um requisição é enviada ao servidor solicitando todas as unidades militares e fortificações inimigas do trecho especificado apenas.

Da mesma forma, quando o servidor necessita notificar os jogadores sobre alguma alteração no ocorrida no estado do jogo este deve fazê-lo de forma otimizada, enviando a notificação apenas para os jogadores que possuem algum elemento de interação posicionado no trecho de mapa cujo estado foi alterado.

5.2.2 Os Elementos de Interação

As fortificações e unidades militares do jogo são importantes elementos do sistema pois é através deles que toda a interação do jogo ocorre - em outras palavras, para que um jogador participe do jogo este deve manipular suas unidades militares e fortificações cujas ações alteram o estado do jogo na porção servidor.

Como citado na seção 5.1, tando as unidades militares quanto as fortificações podem podem ser considerados *sprites*. No entanto, para facilitar a implementação do protótipo esses dois elementos foram separados em duas classes - as unidades militares serão chamadas genericamente de *sprites* e as fortificações de *cidades*, melhor descritos abaixo.

Os *Sprites*

Para o protótipo *Alea Jacta Est* os *sprites* são considerados elementos de interação que se movimentam através do mapa do jogo. O único elemento de interação que possui tal característica são as unidades militares do jogo.

A porção servidor gerencia todas as unidades militares participantes de uma sessão de jogo, independente de quais jogadores as possuam. Devido a essa característica é necessário rotular cada uma dessas unidades para evitar confusão durante os procedimentos que as manipulam, tornando o estado do jogo inconsistente.

Para evitar tal conflito definiu-se que cada unidade militar em jogo deve possuir uma **chave**. Uma chave consiste em uma valor numérico inteiro positivo utilizado para identificar unicamente cada um dos elementos participantes de uma sessão de jogo. Seu valor é incrementado a cada novo elemento criado, tornando impossível reutilizar uma chave.

Além desse atributo todo elemento que representa uma unidade militar possui outras informações associadas, brevemente descritas abaixo:

- `chave`: identifica unicamente cada unidade militar em jogo.
- `idSprite`: utilizado para definir qual é o tipo de unidade militar representada por esse *sprite*. Os tipos disponíveis foram apresentados na Figura 3.3.
- `posX`: armazena a coordenada x da posição da unidade militar no mapa.
- `posY`: armazena a coordenada y da posição da unidade militar no mapa.
- `ataque`: atributo utilizado para definir a probabilidade de sucesso em um confronto iniciado por essa unidade.
- `defesa`: atributo utilizado para reduzir a probabilidade de sucesso no confronto com uma unidade militar atacante.
- `custo`: define quantos turnos uma fortificação deve ficar indisponível quando criando um *sprite* desse tipo.

As Cidades

As fortificações - ou genericamente cidades - são importantes elementos de interação pois é através deles que cada jogador realiza a manutenção de seus exércitos.

Cada jogador pode ter quantas fortificações desejar, sendo que cada uma delas é criada através do "sacrifício" de uma unidade *construtor*.

Cada fortificação em jogo é única. Isso significa que a exemplo dos *sprites* cada fortificação é identificada por uma **chave**. Tal chave é incrementada para cada nova fortificação construída, não utilizando valores já atribuídos a outras fortificações.

Em termos de implementação, uma fortificação ou cidade consiste em um elemento que encapsula as seguintes características:

- `chave`: valor utilizado para identificar unicamente cada cidade em jogo.
- `posX`: corresponde a coordenada x da posição da cidade no mapa do jogo.
- `posY`: armazena o valor da coordenada y da posição da cidade no mapa do jogo.
- `flagEstado`: consiste em um *flag* utilizado para indicar o estado corrente da fortificação. Tal estado pode admitir apenas dois valores: *em construção* ou *fora de construção*.
- `idSprite`: utilizado pela fortificação para determinar qual é a unidade militar a ser construída caso o estado da fortificação seja *em construção*.
- `turnosInativos`: armazena a quantidade de turnos em que a fortificação deve ficar inativa enquanto estiver no estado *em construção*. Após passados essa quantidade de turnos a fortificação constrói a unidade militar solicitada e entra no estado *fora de construção*.

Uma fortificação pode estar em apenas um dos estados acima citados. Quando ela encontra-se no estado *fora de construção*, é possível realizar uma ação para a fortificação quando esta for o elemento ativo do turno. Se a ação do jogador do turno consistir na criação de novas unidades militares a partir dessa fortificação ocorre uma transição do estado *fora de construção* para o estado *em construção*, como indicado na Figura 5.2.

Quando isso ocorre, a fortificação torna-se indisponível durante uma certa quantidade de turnos que é dada pelo atributo **custo** da unidade militar a construir. Nesse estado nenhuma ação pode ser realizada por essa fortificação, que torna-se novamente disponível após passados tantos turnos quanto necessário para a construção da unidade selecionada.

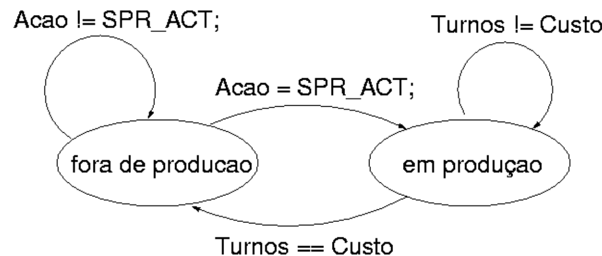

Figura 5.2: Diagrama de estados para uma fortificação

5.2.3 Os Jogadores

Os jogadores consistem no principal elemento manipulado pela porção servidor. Pode ser considerado o verdadeiro "bloco de construção" do sistema, pois é através desse elemento que o sistema tem acesso ao estado do jogo para cada um dos jogadores participantes da sessão.

Cada jogador possui uma quantidade de unidades militares e fortificações que formam seus exércitos e império. É através desses elementos que o jogador interage com o sistema, executando uma ação para cada elemento ativo durante seu turno de jogo.

Em nível de implementação um jogador é uma entidade que agrupa diversos elementos de interação com outras propriedades intrínsecas do mesmo. Os principais atributos que são encapsulados por uma entidade jogador são:

- `chave`: de forma equivalente a um *sprite* ou cidade, a chave identifica unicamente cada jogador pertencente a sessão de jogo.

- `lista de` *sprites*: cada jogador possui uma lista de unidades militares que lhe pertencem. As unidades militares não possuem nenhum atributo que especifica a quem pertencem. A única forma de sabê-lo é verificando a pertinência dessa unidade na lista de *sprites* de cada jogador da sessão.

- `lista de cidades`: de forma semelhante ao atributo `lista de` *sprites*, essa lista armazena todas as fortificações que pertencem a um jogador. Cada fortificação é identificada por uma **chave** e deve ser buscada nessa lista caso seja necessário manipulá-la.

Cada jogador pertencente a uma sessão de jogo em execução pode estar em apenas um dos dois estados ilustrados no diagrama da Figura 5.3. Quando o

mesmo encontrar-se no estado *waiting*, nenhuma ação poderá ser executada por esse jogador, indicando que o mesmo não encontra-se em seu turno. Nesse estado o jogador apenas recebe notificações do servidor informando sobre alterações ocorridas no estado do jogo.

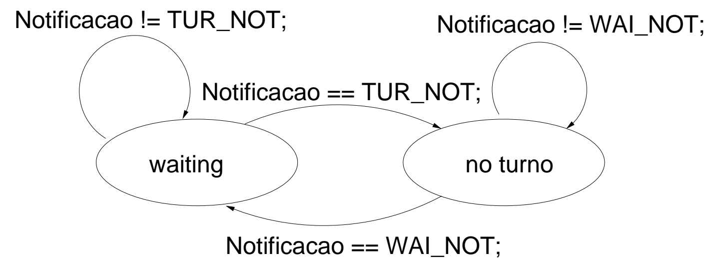

Figura 5.3: Diagrama de estados para um jogador pertencente a uma sessão de jogo em execução

Um jogador que esteja no estado *waiting* realiza uma transição para o estado *no turno* quando recebe do servidor uma notificação informando que o turno de jogo corrente lhe pertence. Quando isso ocorre esse jogador deve realizar uma ação para cada unidade militar em jogo e cada fortificação no estado *fora de produção*.

Para cada tentativa de ação realizada pelo jogador nesses elementos de interação existe uma notificação do servidor informando sobre o sucesso ou fracasso da ação. Quando o jogador encerrar todas as ações permitidas para todos os elementos de interação disponíveis em seu turno este realiza uma transição para o estado *waiting* novamente, indicando que não é mais o jogador do turno corrente.

5.2.4 O gerenciador de sessão

O gerenciador de sessão é o elemento da porção servidor responsável por controlar os demais elementos do jogo. Tal elemento consiste em uma `thread` de execução para cada sessão de jogo criada no servidor. Todas as referências feitas nas seções acima que referem-se a interação ocorrida durante a execução de jogo são controladas por ele, podendo ser considerada a própria porção servidor da sessão.

A função desse elemento é receber mensagens de solicitação de ações oriundas da aplicação cliente do jogador do turno, processar tais ações e retornar o resultado dessas ações para todos os jogadores pertencentes ao trecho de mapa onde ocorreu a mudança no estado do jogo.

A principal estrutura de dados que esse elemento possui é a lista de jogadores. É por meio dessa lista que o gerenciador da sessão tem acesso aos elementos de

interação pertencentes a cada jogador, uma vez que estes encontram-se encapsulados nesse tipo de dados. Também é através da lista de jogadores que o gerenciador decide qual é o jogador do turno corrente, verificando quais jogadores ainda participam da sessão em execução. Ao determinar a quem pertence o turno o gerenciador envia uma mensagem de notificação contendo a primitiva `TUR_NOT` para o jogador selecionado, e aguarda por cada ação do mesmo.

O gerenciador de sessão também verifica se o jogo deve continuar em execução. Enquanto existirem dois ou mais jogadores que estejam em jogo e ainda possuam algum elemento de interação disponível a execução da sessão de jogo continua. Quando sobrar apenas um jogador na sessão este será considerado vencedor da partida. Quando isso ocorre o gerenciador envia mensagens de notificação contendo a primitiva `FIM_NOT` aos jogadores que ainda participam da sessão em execução, indicando se os mesmos venceram ou perderam a partida.

5.3 A lógica do jogo

Nessa seção será apresentado o funcionamento da lógica do jogo *Alea Jacta Est* através de exemplos que ilustam diversas situações comumente encontradas em uma sessão de jogo em andamento. Junto com a descrição de tais situações será descrito como as porções cliente e servidor reagem por meio de diagramas de comunicação.

As próximas seções apresentam todos os tipos de situação possível de ocorrer durante uma sessão de jogo em execução através de figuras e diagramas de comunicação utilizados na troca de mensagem entre as porções cliente e servidor. Tais ações compreendem tentativa de realizar movimento de unidades militares, criação de fortificações, criação de novas unidades militares entre outras.

5.3.1 Movimentando uma unidade militar

Quando o elemento de interação ativo do jogador do turno for uma unidade militar o jogador pode tentar movimentá-la pelo mapa. Como descrito na seção 4.1.4, para que a unidade militar ativa seja movimentada pelo mapa o jogador deve pressionar a tecla numérica correspondente a uma das oito posições adjacentes à posição da unidade militar em questão.

Diante dessa ação diversas situações podem surgir. Cada uma delas será descrito abaixo em detalhes.

Situação 1: O movimento é inválido

A situação de movimento inválido ocorre quando o jogador tenta movimentar uma unidade militar para uma posição do mapa ocupada por um *tile* que representa um obstáculo. Quando isso ocorre a porção servidor do jogo notifica ao jogador que tal movimento é inválido, fazendo com que este permaneça na posição em que ocupava antes da tentativa de movimento.

A Figura 5.4 (a) exibe uma situação onde isso ocorre. Uma unidade militar *construtor* está posicionada acima de um *tile* que representa uma árvore. Este *tile* não permite movimento, logo qualquer tentativa de movimento para a posição ocupada por ele não terá êxito.

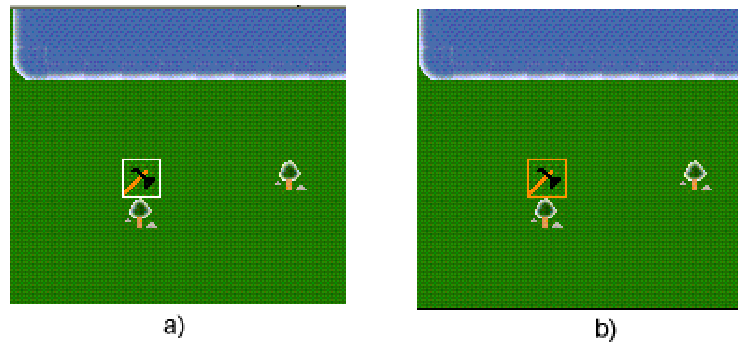

Figura 5.4: Tentativa de movimento inválido. a) O jogador tenta movimentar a unidade *construtor* uma posição abaixo. b) o servidor notifica o cliente que o movimento é inválido, logo a unidade *construtor* não altera sua posição.

A sequência de jogo descrita acima ocorre da seguinte forma: o jogador do turno pressiona a tecla numérica correspondente ao número `8` numa tentativa de realizar movimento da unidade *construtor* para baixo. A aplicação cliente cria uma mensagem contendo a primitiva `MOV_ACT` com a posição da tentativa de movimento e a envia ao servidor, como ilustrado na Figura 5.5 (I). Os parâmetros da mensagem enviada correspondem respectivamente à chave e posição (x, y) do *sprite construtor* que tenta realizar o movimento.

Ao receber essa mensagem, a porção servidor carrega o jogador do turno da lista de jogadores da sessão corrente. O servidor procura na lista de *sprites* do jogador carregado se existe algum *sprite* cuja chave corresponde a chave especificada na mensagem.

O trecho do mapa que contém a posição desejada é carregado, verificando-se

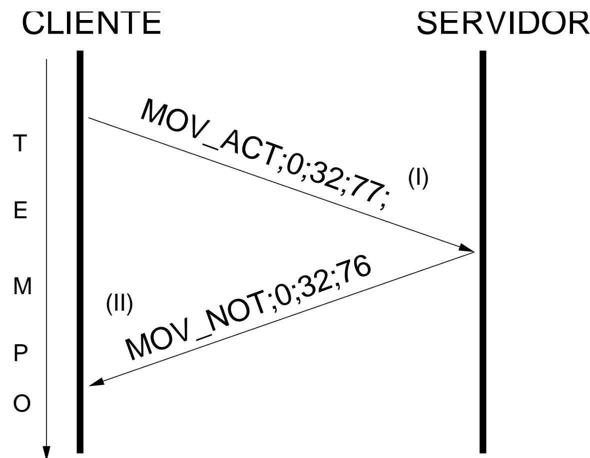

Figura 5.5: Troca de mensagens para um movimento inválido. (I) o cliente envia a requisição de movimento para uma posição inválida. (II) o servidor envia a notificação indicando a posição antiga do *sprite*.

que o *tile* dessa posição não permite movimento. O servidor cria uma mensagem de notificação contendo a primitiva `MOV_NOT` com a posição antiga do *sprite* e a envia para a aplicação cliente do jogador do turno. Tal procedimento é ilustrado na Figura 5.5 (II).

Ao receber essa notificação a porção cliente não modifica a posição do *sprite*, mantendo-o na posição em que estava como mostra a Figura 5.4 (b).

Situação 2: O movimento é válido

Para ilustrar essa possibilidade considere a situação apresentada na Figura 5.6 (a). O jogador do turno deseja movimentar a unidade *construtor* ativa uma casa à direita.

Para realizar tal movimento a seguinte sequência de ações ocorre: a tecla numérica correspondente ao número `6` é pressionada pelo jogador do turno. A aplicação cliente cria uma mensagem contendo a primitiva `MOV_ACT` com os parâmetros da ação de movimento e a envia para o servidor, como ilustrado na Figura 5.6 (I).

A porção servidor, ao receber a mensagem do cliente decodifica-a e realiza um procedimento equivalente ao ocorrido na **Situação 1**. Porém, ao verificar que o *tile* da posição especificada permite movimento este cria uma mensagem contendo a primitiva `MOV_NOT` tendo como parâmetro a nova posição a ocupar, como ilustrado na Figura 5.7 (II).

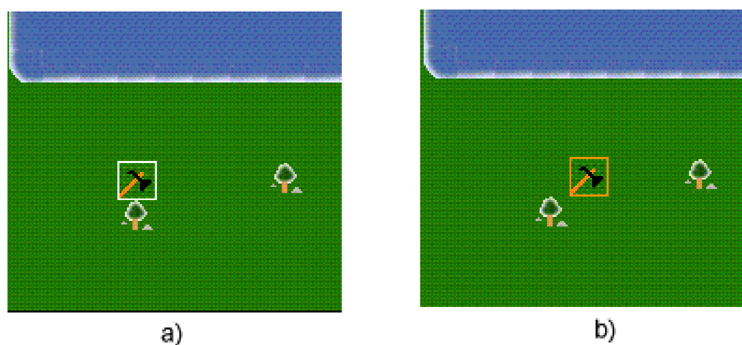

Figura 5.6: Tentativa de movimento válido. a) O jogador tenta movimentar a unidade *construtor* uma posição a direita. b) o servidor notifica o cliente que o movimento é válido e altera a posição desse *sprite* para a posição do movimento solicitado.

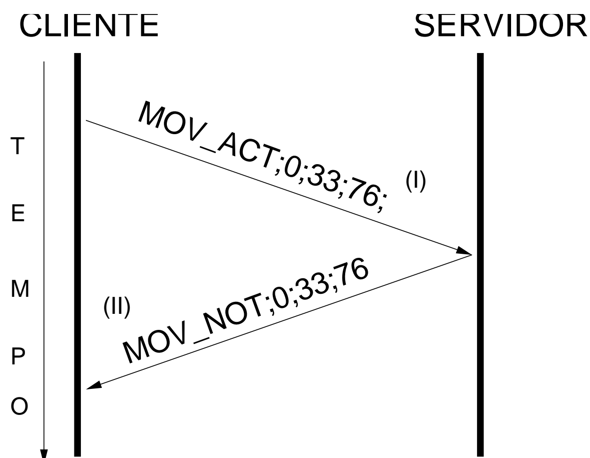

Figura 5.7: Troca de mensagens para um movimento válido. (I) o cliente envia a requisição de movimento para uma posição válida. (II) o servidor verifica no trecho de mapa correspondente a essa posição que a mesma é válida, e envia a notificação indicando a nova posição do *sprite*.

Ao receber tal mensagem a aplicação cliente decodifica-a, efetuando as modificações ocorridas na unidade militar *construtor* ativa. Essa situação é ilustrada na Figura 5.6 (b).

Situação 3: O movimento caracteriza um confronto

Essa situação ocorre quando uma unidade militar ativa tenta realizar um movimento para uma posição que esteja ocupada por outra unidade militar ou fortificação inimiga. Quando isso ocorre esse movimento caracteriza um confronto.

Em um confronto apenas um dos elementos de interação envolvidos sobrevive. No protótipo *Alea Jacta Est* o servidor utiliza um modelo probabilístico simples para determinar a chance de um ataque ser bem sucedido. Tal modelo utiliza dois valores para o cômputo da chance: o atributo **ataque** da unidade militar que solicitou o movimento e o atributo **defesa** do elemento de interação que ocupa a posição cujo movimento foi solicitado.

O modelo probabilístico tenta prever a chance de uma determinada unidade militar obter sucesso em um confronto com outra unidade de interação. Após calculada essa probabilidade, o sistema simula o ataque gerando um número aleatório entre `0` e `1`.

Tal valor é comparado com a probabilidade dada pelo modelo. Caso este valor seja menor ou igual a probabilidade dada o ataque é bem sucedido e a unidade de interação derrotada deve ser removida da lista do jogador que a possui. Caso contrário o ataque é mal sucedido e o *sprite* atacante deve ser destruído.

O modelo probabilístico utilizado no protótipo *Alea Jacta Est* é o dado pela Equação 5.1.

$$Probabilidade = \frac{Ataque}{Ataque + Defesa} \tag{5.1}$$

Como pode ser visto no modelo, a chance de um unidade realizar um ataque bem sucedido é reduzida pelo atributo **defesa** do elemento de interação defensor.

A Figura 5.8 ilustra uma situação envolvendo o confronto entre duas unidades militares. O jogador do turno tenta realizar um movimento com a unidade *catapulta* ativa para a posição ocupada pela unidade *cohort* de um jogador inimigo, como visto na Figura 5.9 (I).

O servidor recebe a mensagem e processa-a, verificando que a posição requerida encontra-se ocupada por uma unidade militar inimiga, caracterizando um confronto. O confronto é decidido pelo modelo probabilístico proposto, com a unidade *catapulta* vencendo o combate. O servidor cria uma notificação informando a destruição da unidade *cohort* e a envia para todos os jogadores do trecho onde ocorre o combate. Essa sequência é ilustrada na Figura 5.9 (II).

Em seguida outra mensagem de notificação é enviada confirmando o movimento. Tal mensagem contém a primitiva `MOV_NOT` com a posição antes ocu-

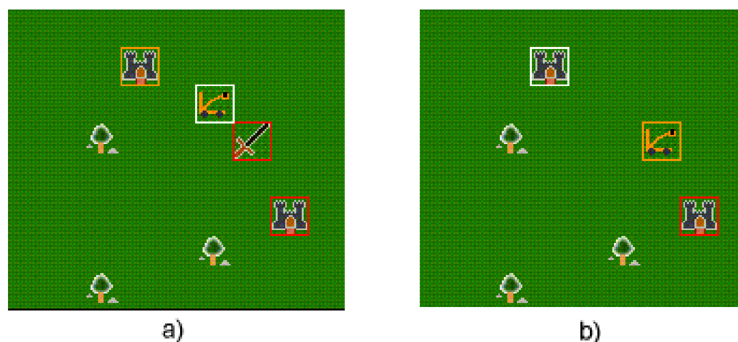

Figura 5.8: Tentativa de movimento que caracteriza um ataque. a) o jogador do turno tenta movimentar a unidade *catapulta* na mesma posição ocupada por um *cohort* inimigo. b) o servidor verifica que a posição está ocupada, realiza o ataque com base nos atributos dos dois *sprites* e notifica aos jogadores de interesse no trecho a destruição da unidade *cohort* e movimento da unidade *catapulta*

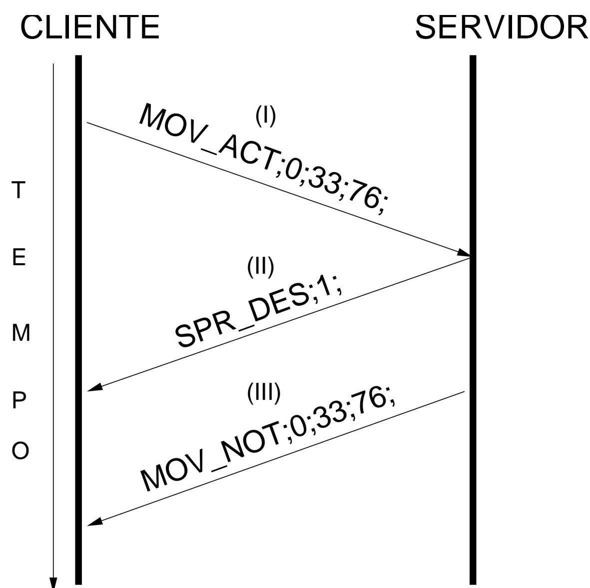

Figura 5.9: Troca de mensagens para um ataque bem sucedido. (I) o cliente envia uma mensagem solicitando um movimento para o servidor. (II) o servidor verifica que a posição cujo movimento foi solicitado encontra-se ocupada e realiza o procedimento de confronto, onde o *sprite* de chave 1 é destruído. (III) Uma notificação informando que o movimento deve ocorrer é enviado novamente ao cliente, que renderiza os resultados.

pada pela unidade militar derrotada. O diagrama de comunicação representado na Figura 5.9 ilustra essa situação em (III). Após receber a notificação a aplicação cliente renderiza o novo estado do trecho ilustrado na Figura 5.8 (b).

A outra possibilidade de resultado para um confronto consiste no fracasso do ataque da unidade militar ativa do jogador do turno. Quando isso ocorre o servidor notifica os jogadores com elementos de interação no trecho do confronto sobre a destruição da unidade atacante através de uma mensagem contendo a primitiva `SPR_DES`. Tal situação é ilustrada na Figura 5.10.

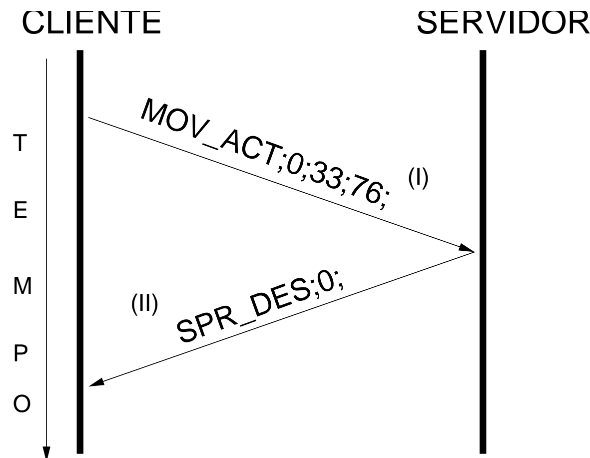

Figura 5.10: Troca de mensagens para um ataque mal sucedido. (I) o cliente envia uma requisição de movimento para o servidor. (II) o servidor verifica que a posição cujo movimento foi solicitado encontra-se ocupada e realiza o procedimento de confronto, onde o *sprite* é destruído.

O procedimento descrito na **Situação 3** é equivalente à situação onde a unidade militar ativa do jogador do turno tenta movimentar-se para uma posição ocupada por uma fortificação. A única diferença entre ambas situações está na mensagem de notificação enviada caso a fortificação seja destruída - a primitiva enviada nessa situação é a `CID_DES`. O restante do procedimento é equivalente, sendo desnecessário exemplificar a situação.

5.3.2 Abstendo-se de realizar uma ação para um elemento de interação

Cada jogador tem o direito de realizar uma ação para cada elemento de interação que pertença a ele durante seu turno de jogo. Porém, essa ação pode corresponder

a uma ação nula - em outras palavras, o jogador decide abster-se de realizar a ação para o elemento de interação ativo. Quando isso ocorre o cliente envia ao servidor uma mensagem indicando que não deseja realizar nenhuma ação, como ilustra a figura5.11 (I). Ao receber tal mensagem o servidor responde com uma notificação indicando que nenhuma ação foi realizada - Figura 5.11 (II) - passando o direito para o próximo elemento de interação ativo.

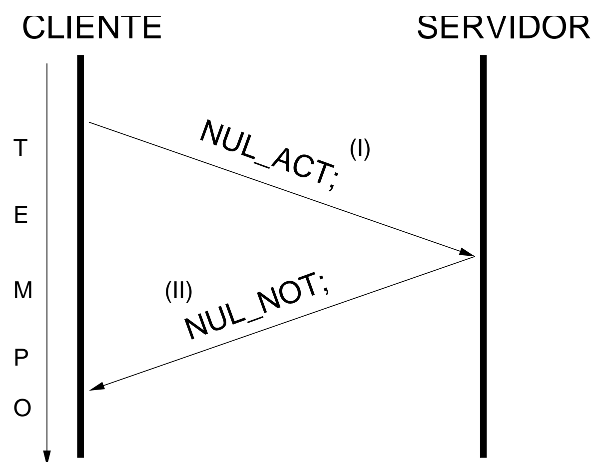

Figura 5.11: Troca de mensagens para uma ação nula. (I) o cliente envia uma mensagem para o servidor indicando que não irá realizar nenhuma ação para o elemento de interação ativo nesse turno. (II) o servidor envia uma mensagem de notificação para o jogador indicando que nenhuma ação foi realizada.

5.3.3 Criando uma fortificação

Quando uma unidade militar *construtor* é a unidade de interação ativa do jogador do turno corrente é possível “sacrificá-la” para criar uma fortificação na posição antes ocupada pelo *construtor*. Quando isso ocorre uma seqüência de mensagens é trocada entre o cliente e o servidor, culminando com a criação de uma fortificação quando possível.

Um exemplo disso é dado pela Figura 5.12. O jogador do turno possui como elemento de interação ativo uma *catapulta*, ilustrada em (a). Ao tentar criar uma fortificação este jogador recebe uma notificação oriunda do servidor informando que tal ação não é válida, uma vez que a unidade de interação ativa não é uma unidade *construtor*. Esse procedimento é ilustrado na íntegra pela Figura 5.13.

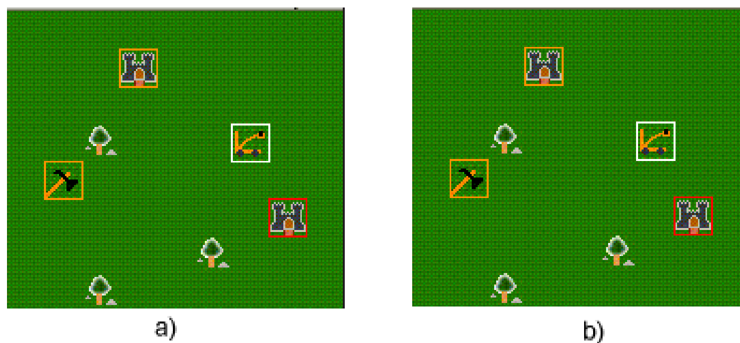

Figura 5.12: Tentativa de criar uma fortificação a partir de uma *sprite* não-*construtor*. a) a solicitação é enviada ao servidor com a chave da unidade *catapulta*. b) o servidor envia uma mensagem de erro indicando ser impossível realizar tal ação.

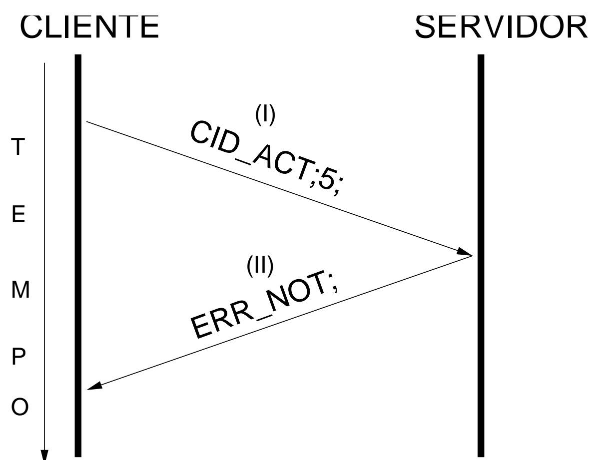

Figura 5.13: Troca de mensagens para uma tentativa mal sucedida de criar uma fortificação. (I) o cliente envia uma mensagem solicitando a criação de uma fortificação a partir do *sprite* que possui a chave enviada com a primitiva `CID_ACT`. (II) o servidor notifica o cliente que é impossível realizar essa ação a partir do *sprite* indicado.

Quando isso ocorre o elemento de interação ativo não perde o direito de ação no turno, devendo realizar qualquer outra ação válida, situação ilustrada na Figura 5.12 (b).

Se o elemento de interação ativo do jogador do turno for uma unidade *construtor* então a sequência de comunicação executada quando o jogador solicita a cons-

trução de uma fortificação será diferente. Essa situação é ilustrada pela Figura 5.14 (a). A aplicação cliente do jogador do turno envia uma mensagem ao servidor contendo a primitiva CID_ACT solicitando a criação de uma fortificação através do "sacrifício" da unidade *construtor* ativa, situação ilustrada na Figura 5.15 (I).

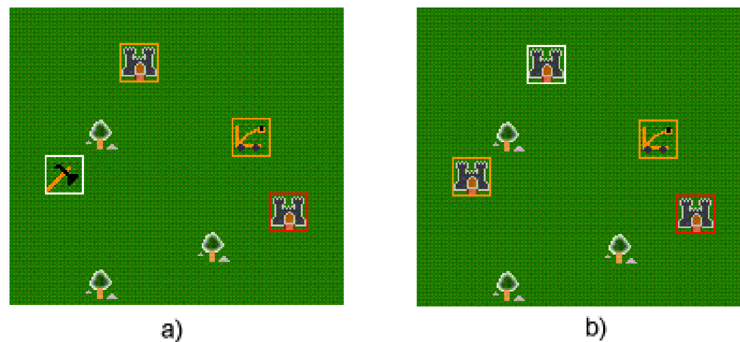

Figura 5.14: Criação de uma fortificação a partir de uma unidade *construtor*. a) uma unidade construtor seleciona a posição a fundar a fortificação. b) após processamento o servidor permite que a aplicação cliente crie uma fortificação na posição indicada.

Ao receber essa mensagem o servidor remove a unidade *construtor* da lista do jogador do turno e envia uma mensagem para os jogadores com elementos de interação no trecho onde ocorre a construção informando a destruição desse *construtor* por meio da primitiva SPR_DES, como ilustrado na Figura 5.15 (II).

Em seguida outra notificação é enviada para tais jogadores informando sobre a criação de uma nova fortificação no jogo, como ilustra a Figura 5.15 (III). Após receber a notificação o cliente renderiza o trecho de mapa com o novo estado do jogo e apresenta no *display* do MID algo semelhante à cena ilustrada na Figura 5.14 (b).

5.3.4 Criando uma unidade militar a partir de uma fortificação

Se o elemento de interação ativo for uma fortificação o jogador do turno corrente tem a possibilidade de criar novas unidades militares a partir dela. Tal procedimento demora alguns turnos para ser completado, sendo que a quantidade de turnos em que a fortificação fica no estado *em contrução* depende do valor do atributo **custo** da unidade militar a construir.

Para ilustrar esse procedimento, a situação indicada na Figura 5.16 é considerada. O elemento de interação ativo do jogador do turno é uma fortiticação (a).

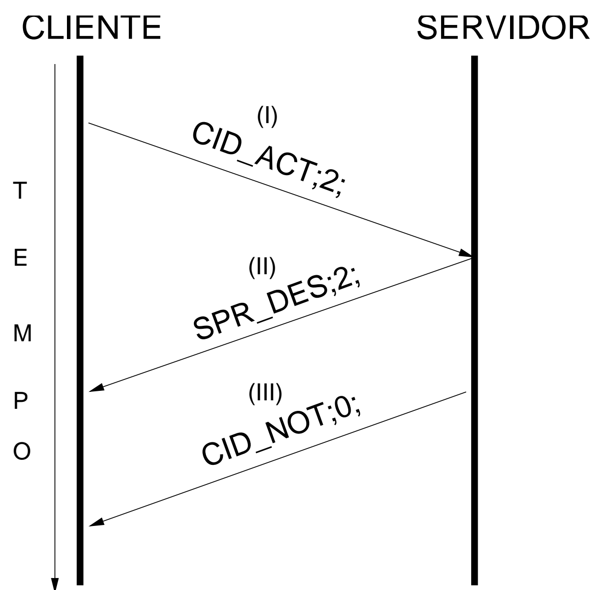

Figura 5.15: Troca de mensagens envolvida na criação de uma fortificação. (I) uma mensagem de requisição é enviada ao servidor. (II) o servidor notifica o cliente sobre o "sacrifício" da unidade *construtor*. (III) O servidor notifica o cliente que uma nova fortificação está em jogo.

Para o jogador construir uma unidade *cavalaria* a partir dessa fortificação este deve pressionar o *soft-button* correspondente ao comando `Criar`.

Tal procedimento faz com que uma lista de *sprites* possíveis de serem construídos seja apresentado ao jogador, que seleciona a unidade desejada (b) e pressiona o *soft-button* correspondente ao comando `Criar`. A aplicação cliente cria uma mensagem contendo a primitiva `SPR_ACT` e a envia para o servidor, como ilustrado na Figura 5.17 (I).

O servidor recebe a mensagem de solicitação e decodifica-a extraindo as informações necessárias para a criação da nova unidade militar. Ao encontrar a fortificação ativa da lista de cidades do jogador do turno o servidor altera o estado da mesma para *em construção*, especificando qual é o tipo de unidade militar a construir. Em seguida uma mensagem de notificação contendo a primitiva `NUL_NOT` é enviada em resposta a requisição do cliente informando que a unidade militar solicitada ainda não está criada.

No fim do turno de cada jogador que possui fortificações no estado *em construção* o servidor envia uma série de mensagens para o jogador do turno, sendo uma mensagem para cada fortificação do jogador nesse estado.

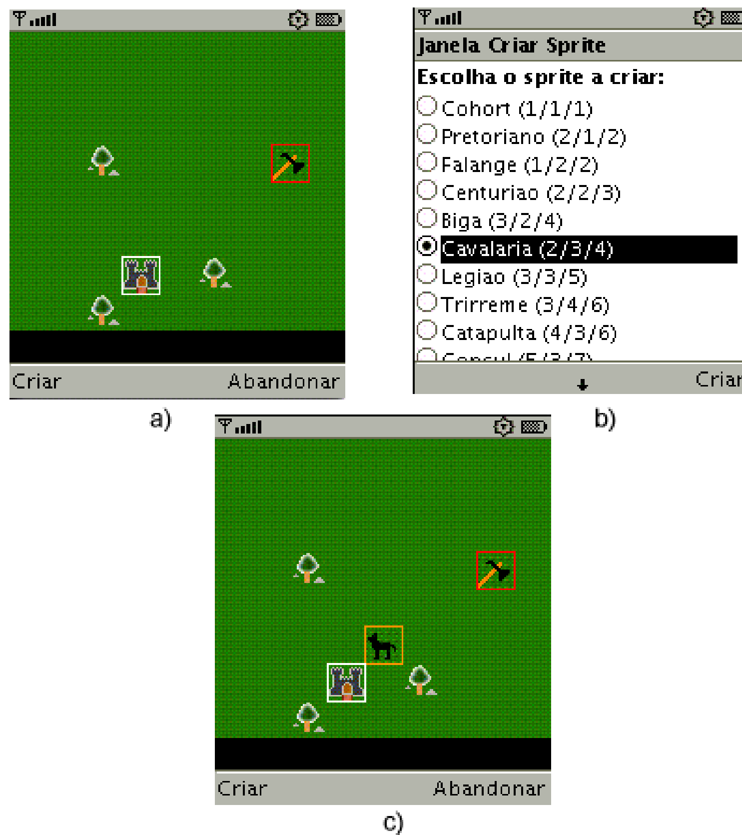

Figura 5.16: Procedimento de criação de novas unidades militares. a) o jogador escolhe criar um *sprite* de uma fortificação que é o elemento de interação ativo do turno. b) o jogador seleciona qual unidade militar criar. c) após passados tantos turnos quanto necessário para a contrução da unidade militar, a fortificaçao torna-se novamente disponível para o jogador.

Quando essa quantidade de turnos é cumprida o jogador do turno corrente recebe uma notificação do servidor contendo a primitiva `SPR_NOT`, como ilustrado na Figura 5.17 (III).

Quando isso ocorre, a porção servidor novamente altera o estado da fortificação para *fora de construção*, permitindo que novas ações sejam executadas por ela. A nova unidade militar criada também torna-se disponível para o jogador, como ilustrado na Figura 5.16 (c).

Essa nova unidade militar é incorporada aos exércitos do jogador do turno. Quaisquer unidades militares podem ser criadas dessa maneira, incluindo novas

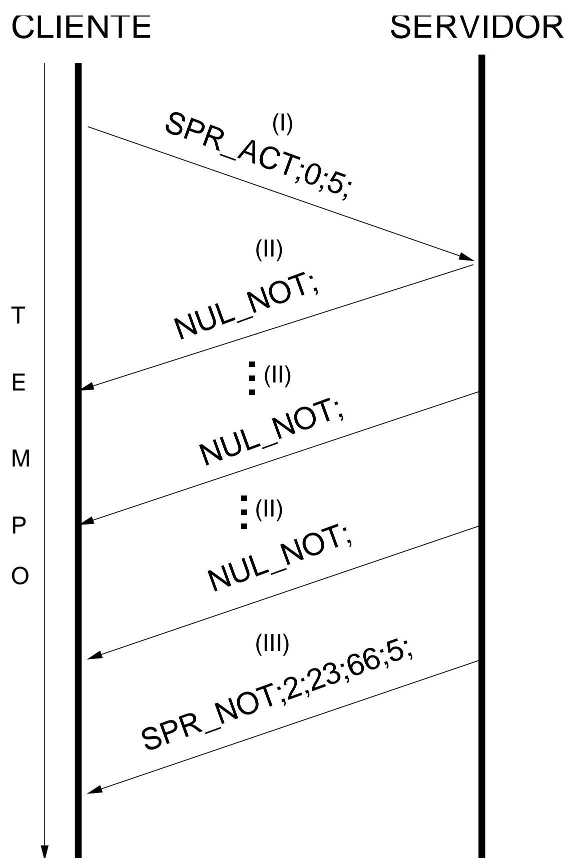

Figura 5.17: Procedimento de criação de novas unidades militares. (I) uma mensagem é enviada do cliente para o servidor solicitando a criação de um *sprite* a partir da fortificação especificada. (II) o servidor envia notificações contendo a primitiva `NUL_NOT` para o jogador corrente no fim de seu turno enquanto a unidade não estiver construída. (III) O servidor notifica o cliente sobre a criação do *sprite* através de uma mensagem que contém a primitiva `SPR_NOT` com a chave do novo *sprite*.

unidades *construtor* futuramente utilizadas para fundação de outras fortificações.

Capítulo 6

Considerações Finais

Este capítulo apresenta uma breve conclusão sobre os resultados obtidos com esse trabalho. Diversas foram as dificuldades encontradas, e diversas são as propostas de melhoria do protótipo. Para melhor apresentar esses resultados o capítulo foi dividido em duas seções. A seção 6.1 descreve os conhecimentos obtidos com a implementação do protótipo, apresentando as vantagens e desvantagens oferecidas pela plataforma de desenvolvimento Java 2, além das facilidades e dificuldades encontradas durante a implementação desse protótipo. A seção 6.2 apresentará novas propostas de trabalhos futuros visando melhorias na aplicação implementada.

6.1 Comentários

A implementação de aplicações para dispositivos que possuem restrições de processamento e memória é uma tarefa árdua e complexa, pois leva em conta a preocupação tanto com a otimização do código escrito - que tem influência na velocidade em que a aplicação é executada - quanto na política de consumo da memória do dispositivo, um dos recursos mais valiosos desses sistemas.

Aplicações de jogos *multiplayer* são comumente encontrados para máquinas *desktop*, que são relativamente robustas se comparadas com dispositivos embarcados. Se para essas máquinas a implementação de aplicações distribuídas constitui um desafio, implementar esse tipo de aplicação em uma arquitetura embarcada é ainda mais desafiador.

Diversos são os fatores que devem ser levados em conta no momento da implementação. Código otimizado, uso restrito de recursos de memória, tratamento de eventos inesperados comumente encontrados em redes móveis e outros problemas

tornam difícil o desenvolvimento de aplicações para essa classe de dispositivos.

A implementação do protótipo *Alea Jacta Est* serviu como uma primeira experiência sobre o desenvolvimento de jogos *multiplayer* para celulares e outros dispositivos móveis *wireless*. Diversas foram as dificuldades encontradas durante o desenvolvimento do projeto, e serão brevemente citadas abaixo:

- Inexistência de um *guideline* para desenvolvimento de jogos para dispositivos móveis, em especial para o desenvolvimento de jogos *multiplayers*. O trabalho realizado foi puramente experimental, uma vez que as metodologias para desenvolvimento dessa categoria de jogos para computadores *desktop* não são adequadas para uso em dispositivos *wireless* que possuem grandes restrições de processamento e armazenamento;

- Dificuldade de implementar os procedimentos de comunicação das porções cliente e servidor, uma vez que os tipos de fluxo (*stream*) e comunicação disponíveis para os pacotes `java.io` e `javax.microedition.io` da edição J2ME são relativamente diferentes dos fluxos e tipos de comunicação implementados nos pacotes `java.io` e `java.net` da edição J2SE;

- Ausência de equipamento *wireless* para a etapa de depuração e testes. Todos os testes realizados foram executados sobre o emulador *Wireless Toolkit* 2.0, que implementa comunicação via *sockets* e procedimento de entrega da aplicação *on the air* (OTA). Isso impede que testes massivos sejam feitos para avaliar quesitos importantes como velocidade de execução da aplicação, retardo de tempo de comunicação e consumo de memória, entre outros.

Da mesma forma, utilizar a edição J2ME da plataforma Java 2 para implementar o protótipo do jogo mostrou-se vantajoso devido aos seguintes motivos:

- Documentação extensa. O próprio site da *Sun Microsystems* [1] oferece um extenso material sobre desenvolvimento de aplicações utilizando a plataforma Java 2. Esse material varia desde especificação técnica sobre a linguagem e máquinas virtuais, descrição *online* das API's da plataforma, alguns *guidelines* para o desenvolvimento de aplicações para algumas das edições disponíveis e exemplos de código completos.

- Facilidade de integração. As edições da plataforma Java 2 foram projetadas de forma a serem facilmente integradas, que permite que uma aplicação

[1] `http://java.sun.com`

desenvolvida em J2ME seja facilmente integrada com aplicações desenvolvidas em J2SE e J2EE.

- Uso de interfaces de baixo nível utilizando APIs da edição J2ME. A interface `Canvas` é considerada uma interface de baixo nível pois o desenvolvedor decide como uma determinada tela deve se apresentar no *display* do MID. Essa é uma característica importante para o desenvolvimento de jogos, uma vez que todas as características do MID que são portáveis podem ser controladas por classes contidas nas APIs que manipulam gráficos no `Canvas`.

- Facilidade de manipular eventos de entrada do dispositivo. A edição J2ME utiliza *listeners* simples que tratam os eventos mais comuns para pressionamento de teclas, movimento de dispositivos de ponteiro e *soft-buttons* associados a comandos da aplicação.

- Possibilidade de utilizar *sockets* para a comunicação cliente-servidor. Embora a especificação MIDP apenas obrigue o fabricante do disposito a implementar comunicação via `HTTP`, a possibilidade de utilizar *sockets* torna possível a implementação de aplicações onde o servidor pode gerar uma notificação para o cliente sem que este tenha feito uma requisição.

  Somente com o protocolo `HTTP` seria impossível realizar tal procedimento, uma vez que este funciona de acordo com a sequência *request-response*.

- Possibilidade de utilizar figuras no formato `.png` (*Portable Network Graphics*) com as interfaces de baixo e alto nível disponíveis com a especificação MIDP. Os tipos de terreno e unidade militares do protótipo utilizam pequenas imagens armazenadas com o pacote da aplicação para o processo de renderização no *display* do MID, evitando desperdício de processamento para geração de tais imagens.

Apesar dessas vantagens, algumas dificuldades foram encontradas no desenvolvimento da interface de interação do jogador durante uma sessão de jogo em execução. Tais problemas devem-se ao fato da implementação dos elementos de baixo nível de um jogo *tile-based* - *sprites* e *tiles* - exigirem mecanismos complexos para controle de colisão e renderização. Essas dificuldades ocorreram devido ao uso do perfil MIDP 1.0. A versão 2.0 desse perfil apresenta em suas APIs classes otimizadas para implementação de jogos nesse estilo, tratando da melhor forma os problemas acima citados.

6.2 Propostas de trabalhos futuros

O protótipo *Alea Jacta Est* foi concebido como um protótipo puramente experimental. Dessa forma, diversas melhorias podem e devem ser implementadas no mesmo de forma a torná-lo viável. Abaixo serão brevemente descritas algumas propostas de melhoria da implementação do protótipo, todas baseadas na realização desse trabalho:

- Utilização de diagramas UML para modelagem formal da aplicação. Esses diagramas devem enfatizar características importantes do jogo, como por exemplo a sequência de comunicação cliente-servidor utilizada em resposta a ações do jogador do turno.

- Utilizar as novas classes da API do perfil MIDP orientadas ao desenvolvimento de jogos para celulares e outros dispositivos móveis. Tais classes são otimizadas para essa classe de aplicação e tratam situações cujas soluções antes eram implementadas pelo desenvolvedor da aplicação, como tratamento de colisão e animação/renderização dos *tiles* do jogo.

- Utilizar estruturas de dados mais eficientes que as listas implementadas nesse trabalho. Tabelas *hash* são melhores e mais eficientes para a recuperação rápida de informações. Existem procedimentos que utilizam massivamente a busca nas listas de unidade militares e fortificações de cada jogador, justificando essa nova escolha para estrutura de dados a implementar. Além disso cada unidade militar e fortificação são identificadas por **chaves**, que podem ser reaproveitadas como *hash-codes*.

- Modificar o formato de algumas mensagens trocadas entre as porções cliente e servidor. Primitivas como `MOV_ACT` especificam informações sobre ações a realizar que deveriam estar disponíveis apenas no servidor para impedir o uso de técnicas de *cheating* e *hacking*. Tais primitivas devem ser substituídas por primitivas mais seguras - como por exemplo utilizar `MOV_UP;17;` ao invés de `MOV_ACT;17;32;25;` a fim de evitar trapaças.

- Melhorar o código da porção servidor para aceitar várias conexões ao mesmo tempo. Na atual implementação um novo cliente pode conectar ao servidor apenas quando este não está processando conexões de outros clientes que ainda não criaram ou entraram em uma sessão de jogo.

- Estender a aplicação cliente para a edição J2SE, implementando clientes para o jogo como aplicações *desktop* ou *applets* executando em *browsers*. Isso permite que jogadores conectem e participem de uma sessão de jogo mesmo utilizando plataformas de *hardware* e *software* heterogêneos.

- Adicionar novos elementos de interação e regras na especificação do jogo. Tais elementos poderão incluir elementos de gerência de recursos para a construção de fortificações e unidades militares, além de novos tipos de unidade com funcionalidades variadas (unidades de transporte marítimo, unidades de comércio, etc).

- Melhorar o modelo probabilístico utilizado para definir o vencedor de um confronto com a adição de novos atributos ou mesmo a definição de um novo modelo.

- Implementação de um melhor tratamento dos eventos comumente encontrados em redes móveis. Tais eventos consistem em situações como "queda" do sistema do dispositivo e perda de comunicação. O uso de *timeouts* pode prevenir problemas de sincronização causados por tais situações.

Referências Bibliográficas

[1] IBiznet 2002, Estatísticas. URL: `http://www.ibiznet.com.br`. Acessado em 26 de novembro de 2002.

[2] Site Oficial da Nokia. URL: `http://www.nokia.com.br`. Acessado em 26 de novembro de 2002.

[3] Data Monitor Market Analysis. URL: `http://datamonitor.com/`. Acessado em 28 de novembro de 2002.

[4] Site Telecom Web. URL: `http://www.telecomweb.com.br`. Acessado em 26 de novembro de 2002.

[5] Assis W. M., *Avaliação da tecnologia J2ME no contexto de desenvolvimento de jogos multiplayer para celulares*. Monografia de Graduação, 2003 - Universidade Federal de Lavras, Minas Gerais.

[6] Pessoa C., Ramalho G., Battaiola A. *wGen: Um framework para desenvolvimento de jogos para dispositivos móveis*. In: *Anais do XXII Congresso da Sociedade Brasileira de Computação - SEMISH*. Florianópolis (SC), 15 a 19 de julho de 2002.

[7] Clements, T. *Makind Sense of Cellular*. URL: `http://wireless.java.sun.com/getstart/articles/radio/`. Acessado em 05 de novembro de 2002.

[8] Gosling, J., Joy, B., Steele, G. *The Java Language Specification*. Addison-Wesley, 1996. URL: `ftp://ftp.javasoft.com/docs/specs/langspec-1.0.pdf/`. Acessado em 5 de junho de 2002.

[9] Java 2 Standard Edition. URL: `http://java.sun.com/j2se/`. Acessado em 02 de novembro de 2002.

[10] Java 2 Enterprise Edition. URL: http://java.sun.com/j2ee/. Acessado em 02 de novembro de 2002.

[11] Java 2 Micro Edition. URL: http://java.sun.com/j2me/. Acessado em 02 de novembro de 2002.

[12] Deitel H. M., Deitel P. J. *Java: Como Programar. Quarta Edição*. Bookman, 2002.

[13] Ortiz C., Guiguére E. *Mobile Information Device Profile for Java 2 Micro Edition*. Willey Computer Publishing, 2001.

[14] White J. P., Hemphill D. A., *Java 2 Micro Edition, Java in Small Things*. Manning Publications, 2002.

[15] Muchow J. W. *Core J2ME Technology & MIDP*. Sun Press, 2002.

[16] *Mobile Information Device Profile*. URL: http://java.sun.com/products/midp/. Acessado em 02 de novembro de 2002.

[17] *Connected Device Configuration (CDC) and the Foudation Profile - Technical White Papers*. URL: http://java.sun.com/products/cdc/wp/CDcwp.pdf. Acessado em 27 de outubro de 2002.

[18] *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*. White papers. URL: http://java.sun.com/products/cldc/wp/KVMwp.pdf. Acessado em 27 de outubro de 2002.

[19] Burnf, E. M. *História da Civilização Ocidental: Do Homem das Cavernas até a Bomba Atômica, 2a. Edição*. Porto Alegre, Editora Globo, 1968.

[20] *MIDP 1.0 Specification (JSR 37)* URL: http://java.sun.com/aboutJava/communityprocess/final/jsr037/index.html. Acessado em 27 de outubro de 2002.

[21] *Over The Air User Initiated Provisioning Recommended Pratices for the Mobile Information Device Profile*. URL: http://java.sun.com/products/midp/OTAProvisioning-1.0.pdf. Acessado em 05 de junho de 2003.

[22] Fiedler S., Wallner M, Weber M. *A Communication Architecture for Massive Multiplayer Games*. URL: http://delivery.acm.

org/10.1145/570000/566503/p14-fiedler.pdf?key1=566503&key2=6475919301&coll=portal&dl=ACM&CFID=11111111&CFTOKEN=2222222. Acessado em 06 de junho de 2003.

[23] Funkhouser T. *Network Topologies for Scalable Multi-User Virtual Environments*. URL: www.bell-labs.com/user/funk/vrais96.ps.gz. Acessado em 06 de junho de 2003.

[24] Smed J., Kaukoranta T., Hakonen H. *Aspects of Networking in Multiplayer Computer Games*. URL: www.tucs.fi/Publications/techreports/TR454.ps.gz. Acessado em 6 de junho de 2003.

[25] *CLDC Specification 1.0* URL: http://java.sun.com/aboutJava/communityprocess/final/jsr030/index.html. Acessado em 05 de junho de 2003.

[26] Sabadello, M. *Small group multiplayer game*. URL: www.cg.tuwien.ac.at/courses/Seminar/SS2001/multiplayer/small_group_multiplayer.pdf. Acessado em 06 de junho de 2003.

[27] Cronin E., Filstrup B., Kurc A. *A Distributed Multiplayer Game Server System*. URL: www.eecs.umich.edu/~bfilstru/quakefinal.pdf. Acessado em 06 de junho de 2003.

[28] Taylor G., *Tile-Based Games FAQ, version 1.2*. URL: ftp://x2ftp.oulu.fi/pub/msdos/programming/docs/tilefaq.12. Acessado em 06 de junho de 2003.

[29] MacIntosh J. *Tile Graphics Techniques 1.0*. URL: ftp://x2ftp.oulu.fi/pub/msdos/programming/docs/tiletech.zip. Acessado em 06 de junho de 2003.

Apêndice A

Formato de Mensagens

A.1 Mensagens de gerência de sessão de jogo

```
NOV_ACT;<LOGIN>;

SES_ACT;<CHAVE_SESSAO>;<LOGIN>;

SES_NOT;

LIS_ACT;

LIS_NOT;<CHAVE_SESSAO>;<LOGIN>;<TOTAL_JOGADORES>; ...  ;

EST_ACT;

EST_NOT;<TOTAL>;<JOG_1>;<JOG_2>; ...  ;<JOG_N>;

COM_ACT;

COM_NOT;

WAI_NOT;

TUR_NOT;

SAI_ACT;

FIM_NOT;VENCEDOR;

FIM_NOT;PERDEDOR;
```

A.2 Mensagens de Ação

```
MOV_ACT;<CHAVE_SPRITE>;<POS_X>;<POS_Y>;

MOV_NOT;<CHAVE_SPRITE>;<POS_X>;<POS_y>;

SPR_DES;<CHAVE_SPRITE>;

MAP_ACT;<ID_TRECHO>;

MAP_NOT;<TILE_1>;<TILE_2>;<TILE_3>; ...  ;<TILE_N>;

SPR_ACT;<CHAVE_CIDADE>;<ID_SPRITE>;

SPR_NOT;<CHAVE_SPRITE>;<POS_X>;<POS_Y>;<ID_SPRITE>;

INI_ACT;<ID_TRECHO>;

INI_NOT;<CHAVE_SPRITE>;<POS_X_1>;<POS_Y_1>;<ID_SPRITE_1>;
    ...  ;

CID_ACT;<CHAVE_SPRITE>;

CID_NOT;<CHAVE_CIDADE>;

CIN_ACT;<ID_TRECHO>;

CIN_NOT;<CHAVE_CIDADE_1>;<POS_X_1>;<POS_Y_1>; ...  ;

CID_DES;<CHAVE_CIDADE>;

NUL_ACT;

NUL_NOT;

ERR_NOT;
```