UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Suporte ao desenvolvimento de jogos multi-jogador, sem exigência de tempo real

Leonardo de Souza Brasil

FLORIANÓPOLIS
2007

UNIVERSIDADE FEDERAL DE SANTA CATARINA

CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Suporte ao desenvolvimento de jogos multi-jogador, sem exigência de tempo real

Leonardo de Souza Brasil

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação

Banca Examinadora:

Prof.º Frank Augusto Siqueira
Prof.º Leandro José Komosinski
Prof.º Ricardo Pereira e Silva

Prof.º Ricardo Pereira e Silva
Orientador

FLORIANÓPOLIS
2007

Leonardo de Souza Brasil

SUPORTE AO DESENVOLVIMENTO DE JOGOS MULTI-JOGADOR, SEM EXIGÊNCIA DE TEMPO REAL

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Orientador: _____
Prof. Ricardo Pereira e Silva

Banca examinadora

_____
Prof. Frank Augusto Siqueira

_____
Prof. Leandro José Komosinski

"E ser-lhe-á dado conhecer o
segredo que vai transformar
seus sonhos em realidade".
Autor desconhecido

Agradecimentos

Agradeço a meus amigos Gilberto, Jonathan e Andrey, por compartilharem seus conhecimentos e pela amizade durante todo o curso.

Agradeço a minha família, meu pai Sidnei, minha mãe Eliete e minha irmã Fernanda, por acreditarem em mim, pelo apoio e carinho em todos os momentos da minha vida, incluindo esses 4 anos de faculdade.

Agradeço a minha namorada Bianca, pela motivação, companhia e amor que são tão importantes em minha vida.

Meu agradecimento especial ao professor Ricardo Pereira e Silva, meu orientador, pela idealização do presente trabalho. Por sua paciência, por ter acreditado e me apoiado durante toda a realização do mesmo.

Agradeço principalmente a Deus, por ter sempre me indicado o caminho certo a seguir e me dar tudo que eu sempre preciso para continuar caminhado.

Resumo

Jogos multi-jogador (multi-player) são jogos de computador que permitem que vários jogadores participem de uma mesma partida em rede. Os jogadores estabelecem uma conexão com um servidor e a partir desse instante podem realizar partidas contra jogadores que estão jogando o mesmo jogo em outra(s) máquina(s).

O desenvolvimento de um jogo multi-jogador apresenta diversas características em comum. Para jogos sem exigência de tempo real e onde não é necessário validar o estado do jogo no servidor, é possível construir um modelo genérico que possibilite a conexão entre jogadores. Dessa forma, é possível construir artefatos reusáveis de software que dêem suporte ao desenvolvimento desses jogos.

A implementação descrita no presente trabalho constitui-se em dois artefatos de software que possibilitam o suporte a jogos sem a exigência de tempo real: um servidor multi-jogos, em que diferentes jogos podem se cadastrar e fornecer a seus jogadores um ambiente multi-jogador, e um framework orientado a objetos, que permite o acesso dos jogos ao servidor e facilita o desenvolvimento de novos jogos.

Abstract

Sumário

Lista de Abreviaturas

OO – Object Oriented (Orientação a Objeto)

GUI - Graphical User Interface (Interface Gráfica com o Usuário)

UML - Unified Modelling Language (Linguagem de Modelagem Unificada)

IDE - Integrated Development Environment (Ambiente de Desenvolvimento Integrado)

RPC – Remote Procedure Call (Chamada de Procedimento Remoto)

RMI – Remote Method Invocation (Invocação Remota de Métodos)

JVM – Java Virtual Machine (Máquina Virtual Java)

API – Application Programming Interface (Interface para Programação de Aplicativos)

IP – Internet Protocol

TCP – Transmission Control Protocol (Protocolo de Controle de Transmissão)

UDP – Universal Datagrama Protocol (Protocolo Universal de Datagrama)

I/O – Input/Output (Entrada/Saída)

NRT – Non Real Time (Não tempo real)

Lista de Figuras

1. Introdução

Os jogos estão presentes na história da humanidade desde os seus primórdios. Com a evolução dos computadores e a criação das primeiras redes, com destaque para a internet, foi possível simular os jogos através de computadores em ambientes virtuais onde jogadores mesmo distantes fisicamente, compartilham do mesmo tempo e espaço "virtuais". Surge nesse momento o conceito de jogos distribuídos ou jogos multi-jogador.

Jogos multi-jogador são jogos de computador que permitem a participação simultânea de vários jogadores em uma mesma partida. Com a computação distribuída os jogos começaram a ser jogados por diversas pessoas ao mesmo tempo, estando elas em qualquer parte do mundo.

Nesses jogos multi-jogador, é possível identificar os processos utilizados para o estabelecimento de uma partida: O usuário (jogador) inicia seu jogo com o intuito de jogar via rede. Ele, então, localiza um servidor onde seu jogo esteja rodando e solicita uma conexão nesse servidor. O jogador é, então, autorizado e pode iniciar partidas com outras pessoas que também estejam conectadas. O jogador realiza operações que são enviadas para os outros jogadores, de forma que seus jogos sempre possuam o estado atual do jogo. O objetivo principal do servidor é conectar os jogadores e possibilitar a troca de informações a respeito das partidas.

Dentro desse processo é possível identificar aspectos que são comuns a diversos jogos multi-jogador. A necessidade de se conectar no servidor, iniciar uma partida, trocar informações com outros jogadores e o principal deles, a existência de um servidor. Os únicos aspectos que diferem dizem respeito à forma como o jogo é implementado, com suas regras e operações específicas.

Seguindo as melhores práticas de Engenharia de Software, o objetivo deste trabalho foi desenvolver um servidor multi-jogos para jogos multi-jogador, levando em conta principalmente o reuso de software. Esse servidor é genérico, no sentido de poder realizar a comunicação via rede[1] entre jogos de diferentes desenvolvedores.

[1] A comunicação via rede pode ser tanto uma conexão de rede local ou uma comunicação através da internet, o único requisito é que o protocolo de rede utilizado seja o IP e o protocolo de transporte o TCP.

Para que um jogo qualquer possa se conectar ao servidor e para facilitar o desenvolvimento de jogos para esse servidor, foi desenvolvido um framework orientado a objetos a ser utilizado pelo desenvolvedor do jogo.

1.1 Motivação

Jogos promovem interatividade, divertimento e aprendizado. Jogos podem envolver somente uma pessoa, também chamados de jogos "*stand-alone"*, mas se tornam ainda mais divertidos quando abrangem mais de uma pessoa e uma disputa pela vitória.

Jogos de computador proporcionam a criação de mundos novos. Cada jogador é inserido em um mundo virtual e pode fazer coisas nunca antes imaginadas.

As disciplinas de Análise e Projeto Orientados a Objetos ministradas pelo professor Ricardo Pereira e Silva têm como projeto final a criação de um jogo de computador. Esses jogos envolvem normalmente a interação entre dois ou mais jogadores. As características principais desses jogos são: eles não são em tempo real e são implementados na plataforma JAVA. Permitir que esses jogos sejam multi-jogador irá proporcionar aos alunos dessas disciplinas, ainda não familiarizados com os conceitos de computação distribuída, um elemento de motivação a mais para o desenvolvimento de bons projetos. Os jogos produzidos pelos alunos, usuários do servidor e do framework, permitirão que jogadores em diferentes máquinas possam realizar partidas nos jogos desenvolvidos.

1.2 Objetivos

A proposta desse trabalho foi a construção de um servidor multi-jogos. Esse servidor não precisaria conhecer a lógica dos jogos que está tratando e sua função principal é sincronizar os jogadores em uma mesma partida (abordagem *fat client*).

Visou-se também construir um framework que permita o acesso ao servidor por parte dos jogos e possa ser usado facilmente pelos desenvolvedores de jogos multi-jogador sem exigência tempo real. O objetivo desse framework é dispensar os desenvolvedores dos detalhes da comunicação via rede, permitindo que sua atenção esteja focada no jogo em si, como se fosse uma aplicação *stand-alone*.

1.3 Organização do Texto

O trabalho está organizado em nove capítulos. O capítulo 2 trata os temas *jogos* e arquitetura cliente/servidor para jogos multi-jogador. O capítulo 3 apresenta as tecnologias Java, estudadas para a construção dos artefatos de software. O capítulo 4 descreve o protocolo de interação entre os artefatos construídos. O capítulo 5 apresenta o servidor multi-jogos. O capítulo 6 apresenta o framework desenvolvido. O capítulo 7 apresenta os procedimentos para adaptação de um jogo ao framework, usando como exemplo um jogo da velha. No capítulo 8 está a conclusão e no capítulo 9 estão as referências bibliográficas. Os anexo 1 e 2 possuem o glossário e casos de uso levantados na construção dos artefatos.

2. Jogos multi-jogador e a arquitetura cliente/servidor

Os primeiros “jogos multi-jogador” ou jogos distribuídos surgiram da necessidade de simulações militares. Nessas simulações existiam basicamente dois problemas: o custo de realização era alto, e existia o risco de se perderem vidas durante as simulações[CEC 2004].

A partir do surgimento da internet os jogos começaram a evoluir e surgiram os primeiro jogos onde era possível jogar através de uma rede local com outros jogadores, estando entre eles o DOOM em 1993. DOOM é um jogo em primeira pessoa na qual pode-se jogar com até 4 jogadores simultaneamente [CEC 2004].

Em 1996, foi criado o jogo Quake. Quake é um jogo de primeira pessoa em tempo-real em um mundo 3-D. Nele é possível jogar através da internet com 16 jogadores simultaneamente e possui uma arquitetura cliente-servidor[CEC 2004].

Esses jogos possuem em comum a necessidade do jogador se conectar a um servidor, onde normalmente já estão conectados outros jogadores esperando para jogar. A partir daí é criada uma partida e os jogadores podem trocar informações de forma atualizar o estado do(s) outro(s) jogador(es).

Essa arquitetura, onde um ou mais clientes se conectam a um servidor (figura 2.1) e passam a trocar informações através do servidor é conhecida como arquitetura cliente/servidor.

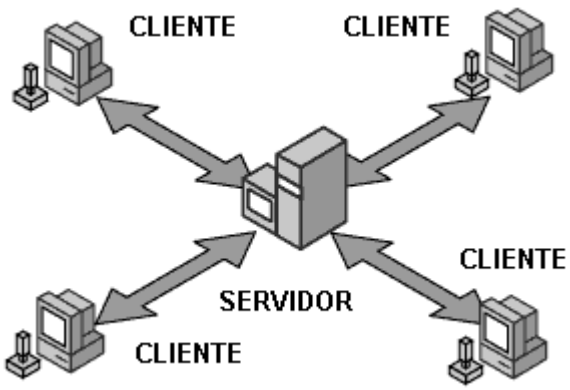

Figura 2.1 Arquitetura Cliente/Servidor. Adaptado de [JOR 2006]

São características desse tipo de arquitetura[ORF 1996] :

- Serviço: Existem dois tipos diferentes de papel em uma arquitetura cliente servidor. De um lado existe o provedor de um ou mais serviços, chamado servidor e do outro o consumidor desses serviços, chamado de cliente. Servidor e Cliente são processos que geralmente rodam em máquinas distintas.

- Recurso Compartilhado: Os recursos são compartilhados no servidor. Além de prover esses serviços para diversos clientes, o servidor pode também regular o acesso a eles.

- Protocolos assimétricos: O servidor é um ator passivo na solicitação de um serviço. O cliente inicia o diálogo com o servidor.

- Trocas baseadas em mensagem: Cliente e servidor são componentes desacoplados. A conversação é realizada através da troca de mensagens. Mensagem é o mecanismo de entrega de solicitações de serviços e suas respostas.

- Encapsulamento de serviços: O servidor provê uma interface de serviços para o cliente. O cliente apenas solicita o serviço. O servidor encapsula o serviço e pode modificar a implementação do serviço sem gerar impacto no cliente, desde de que a interface seja mantida inalterada.

- Escalabilidade: Deve ser possível aumentar o número de clientes conectados ao servidor com um pequeno impacto no desempenho.

- Portabilidade: O servidor pode ser migrado para diferentes máquinas sem dificuldades.

- Integridade: Código e dados do servidor são mantidos de forma centralizada. Os dados compartilhados são íntegros.

Além dos exemplos citados anteriormente pode-se citar também os jogos de rede disponibilizados pelo sistema operacional Windows XP. Além dos jogos *stand-alone* é também possível encontrar algumas versões para jogar com outros usuários conectados no servidor de jogos da Microsoft, são eles: copas, damas, espada, gamão e reversi.

Esse tipo de servidor são constituídos por plataformas voltadas a um jogo ou a um conjunto específico de jogos, sem a possibilidade do usuário implantar novos jogos. Quando um usuário inicia os o jogo em rede e se conecta ao servidor, é criada uma partida do jogo no servidor. Essa partida, assim como a de cada jogador guarda os estados atuais do jogo, permitindo um acesso global e dificultando o uso de trapaças.

2.1 Tipo de Jogos

Os jogos podem ser classificados basicamente em quatro tipos [JOR 2006]:

- FPS (Firs Person Shooters): Jogos em tiro primeira pessoa. Exemplos: DOOM, Quake. Nesses jogos o jogador possui armas de tiro e o objetivo principal é matar seus inimigos para não ser morto pelos mesmos.
- RTS(Real Time Strategy): Jogos de estratégia em tempo real. Exemplo: Age of Empires. Nesse tipo de jogo, que pode ser jogado contra o computador ou conectado à rede, o principal objetivo é organizar uma estratégia que venha a ser melhor que a do adversário. Cada jogador possui uma "comunidade" inicial que evolui conforme passa o tempo, até o ponto que é necessário duelar e se aliar a outros jogadores para conquistar o território total do jogo.
- MMORPG(Massively Multiplayer Online Role Playing Game) – Normalmente traduzido como "jogo de interpretação on-line em massa para múltiplos jogadores", nesse tipo de jogo cada jogador cria um personagem

em mundo virtual e dinâmico. Esse personagem, dependendo do jogo, pode evoluir e para isso precisa realizar tarefas. Nesse jogo milhares de jogadores são conectados e o jogo disponibilizar diversas opções para cada jogador, como duelar com outros jogadores, etc. Exemplos: Second life, LineAge.

- NRT (Non Real Time): Jogos sem exigência de tempo real, diferente dos três jogos anteriormente citados, onde o tempo para responder um evento é definido, esse tempo pode ser alto, mas é sempre definido a priori. Esse tipo de jogo é dirigido por eventos que não tem um tempo definido para acontecer. Exemplos: xadrez, jogo da velha, dama, gamão ou truco.

2.1.1. Jogos sem exigência de tempo real (NRT)

O servidor construído e aqui descrito é voltado para jogos de não tempo real. Isso porque, os jogos que já vinham sendo desenvolvidos pelos alunos da disciplina de Análise e Projeto Orientado a Objetos, foram classificados dentro dessa categoria. Sistemas NRT (Non Real Time) têm como principal característica a não delimitação dos tempos de resposta quando na resposta de um evento. Quando um jogador de xadrez efetua uma jogada, não se sabe a princípio quando o outro jogador ou o computador efetuará a jogada. Esse tempo pode ser próximo a zero(no caso dos computadores), pode demorar horas, ou pode expirar, caso um tempo limite de resposta (*timeout*), seja previamente definido.

Não existe a noção do tempo enquanto se está jogando, diferentemente, por exemplo, de um jogo RTS, onde conforme o passar do tempo eventos são disparados pelo jogo.

Outra característica dos jogos desenvolvidos pelos alunos é o seqüenciamento das jogadas. Cada jogo possui uma forma de organizar os seus jogadores em uma determinada ordem. Um jogador está, em um determinado instante do tempo, jogando ou esperando um outro jogador efetuar sua jogada.

3. Computação Distribuída em Java

Uma das características dos jogos desenvolvidos na disciplina de Analise e Projeto Orientados à Objetos é que eles devem ser desenvolvidos na plataforma Java. Isso significa que os jogos devem rodar "em cima" de uma Java Virtual Machine (JVM). Dessa forma os jogos podem rodar em qualquer plataforma, desde que ela tenha uma JVM instalada.

Por facilidade de desenvolvimento e para que o servidor também possuísse características como portabilidade, optou-se pelo desenvolvimento de todos os artefatos do projeto na plataforma Java. Para tanto se realizou um estudo das tecnologias Java que possibilitassem a comunicação através rede. Iniciou-se pelo estudo de Sockets, componente básico de troca de mensagens em Java, em seguida passou-se a estudar tecnologias de mais alto nível, que proporcionavam chamadas de método remotos: Java RMI e DualRPC. Para implementação decidiu-se pelo uso de DualRPC.

3.1 Sockets

Para que dois processos de software possam se comunicar é preciso antes de tudo criar uma conexão. Uma conexão é um relacionamento entre dois processos, onde dois sistemas sabem da existência um do outro [SIE 2005]. Esse relacionamento permite que bits sejam enviados de um processo para outro por meio de uma conexão física.

Java provê uma API que "esconde" do programador todos os detalhes de baixo nível. Essa API é chamada de *Java Socket API* e a maioria de suas classes e interfaces se encontra no pacote j*ava.net* [AVO 2007].

Quando uma conexão é criada entre um cliente e um servidor, existe aberto entre ambos um canal de dados, que permite que dados binários sejam enviados do cliente para o servidor. Esse canal é um stream da API Java I/O[2].

[2] Para mais detalhes sobre a api Java I/O. Ver em [CAY 2004a] , [SIE 2005] e [AVO 2007].

3.1.1 Funcionamento

Java provê basicamente dois tipos de sockets: *stream sockets* e *datagram sockets* , que usam os protocolos TCP e UDP respectivamente, ambos funcionando sobre o protocolo IP [AVO 2007]. Um socket é simplesmente um artefato construído que representa um ponto final de uma conexão [DEI 2004].

As diferenças entre os stream sockets e os datagram sockets são aquelas advindas dos seus protocolos de comunicação. O modo orientado a conexão, o TCP/IP, oferece serviços mais confiáveis. Existe a garantia que os dados não serão perdidos durante uma comunicação e que a ordem dos pacotes será mantida, ao contrário do UDP/IP, onde mensagens podem ser perdidas e a ordem não é garantida[NUM 2003].

Para garantir a ordem e integridades dos pacotes no protocolo, existe um preço a ser pago: esse protocolo, quando comparado com o UDP, é mais lento, isto é, as mensagens demoram um pouco mais para chegar ao seu destino [NUM 2003].

A próxima sessão trata os sockets TCP/IP, presentes no framework utilizado no trabalho, o DualRPC. Para mais informações sobre os sockets UDP consultar [NUM 2003].

3.1.2 Sockets TCP/IP

Como mostrado na figura 3.1, um aplicativo em socket é constituído de duas entidades, um cliente e um servidor. Ambos possuem um endereço IP, que endereça a máquina onde o aplicativo está rodando na internet, e no caso do servidor, uma porta onde esse aplicativo está registrado.

O servidor assim que instanciado escolhe uma porta e fica aguardando conexões de clientes nessa porta. O cliente estabelece uma conexão com o servidor na porta que está esperando conexões. Caso não ocorram problemas, o servidor aceita a conexão do cliente e cria um socket em uma outra porta escolhida aleatoriamente. Dessa forma, é criado um canal entre cliente e servidor [NUM 2003].

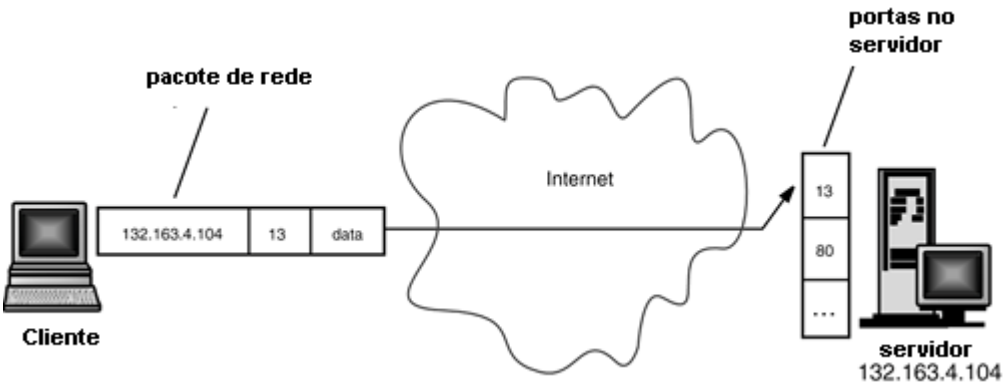

Figura 3.1 Socket Um cliente conectando em uma porta do servidor.
Adaptado de [HOR 2004ª]

Com a conexão estabelecida o cliente pode obter o canal criado com o cliente e enviar informações. Isso é efetuado através da API I/O de Java.

3.2 Java Remote Method Invocation (RMI)

RMI (Remote Method Invocation) é uma das abordagens da tecnologia Java para prover as funcionalidades de uma plataforma de objetos distribuídos e possibilitar chamada de procedimentos remotos (RPC).

RPC é um tipo de protocolo para chamada de procedimentos que se encontram rodando em um outro processo. Diferente de sockets, onde dados são transmitidos entre duas máquinas, usando RPC o cliente pode chamar métodos que se encontram em outro processo como se fosse uma chamada local, exceto pelo fato de que as chamadas acontecem sobre uma rede.[DON 2007].

RMI proporciona o uso de RPC em uma abordagem OO (Orientada a Objetos). O desenvolvedor consegue invocar métodos de objetos que estão presentes em JVM diferentes.

RMI faz parte do núcleo básico de Java desde a versão JDK 1.1, com sua API sendo especificada através do pacote *java.rmi*.

3.2.1 Funcionamento

Em [SUN 2007] é encontrada uma extensa documentação sobre RMI, pretende-se aqui introduzir os principais conceitos de RMI, que foi a primeira alternativa selecionada para a construção do trabalho.

RMI trabalha com uma arquitetura cliente/servidor, ou seja, existem basicamente dois atores, o programa cliente que deseja realizar uma chamada a um outro aplicativo remoto, esse irá processar a chamada e é chamado de servidor [AVO 2007]. (Figura 3.2)

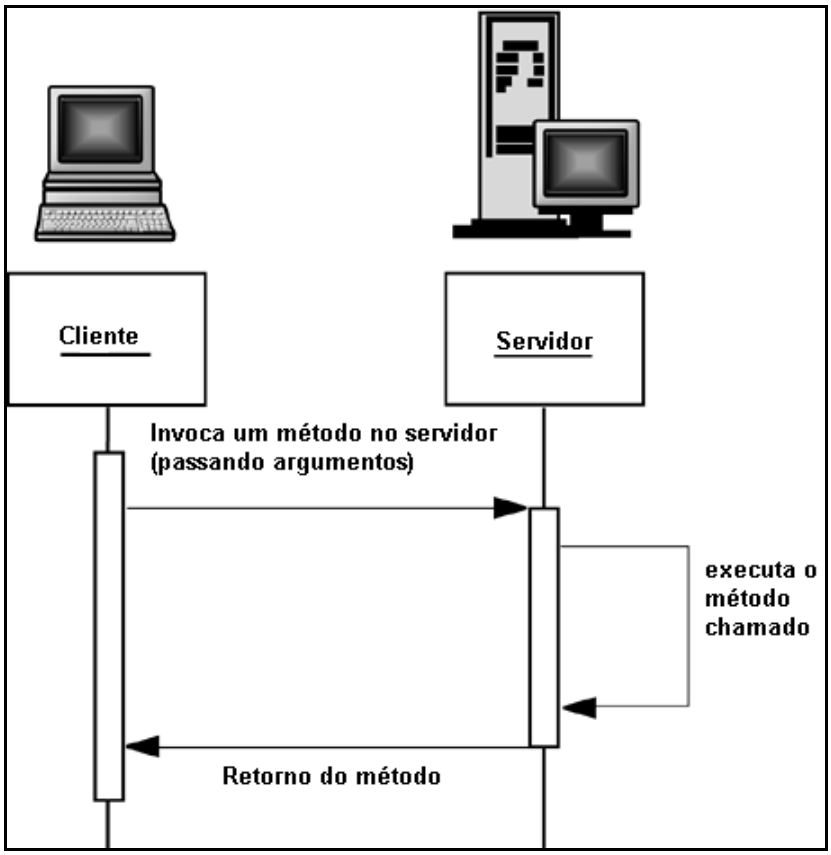

Figura 3.2 Invocando método no servidor. Adaptado de [HOR 2004b]

Para que a comunicação se realize, RMI fornece objetos "auxiliares" que funcionam como *proxies*[3] e intermedeiam a comunicação entre o objeto que realiza a chamada e o objeto que a recebe, executa a funcionalidade e retorna um valor. Dessa forma, o desenvolvedor é liberado de se preocupar com todo o código de baixo nível envolvido nessa troca de informações [SIE 2005].

[3] Padrão de Projeto Proxy, ver em [GAM 1995]

Do lado do cliente, existe um auxiliar que captura as chamadas realizadas pelo cliente e envia essas chamadas para o servidor. Esse auxiliar é chamado de Stub. O servidor, por sua vez, também possui um auxiliar, chamado Skeleton, que captura a chamada passada por um Stub e chama o método no servidor. Se o método possuir um retorno, o trajeto ao inverso é realizado, o Skeleton passa ao Stub o valor retornado (Figura 3.3).

Com esse esquema de objetos "auxiliares" o RMI realiza a invocação a objetos remotos, simulando-os na máquina do cliente.

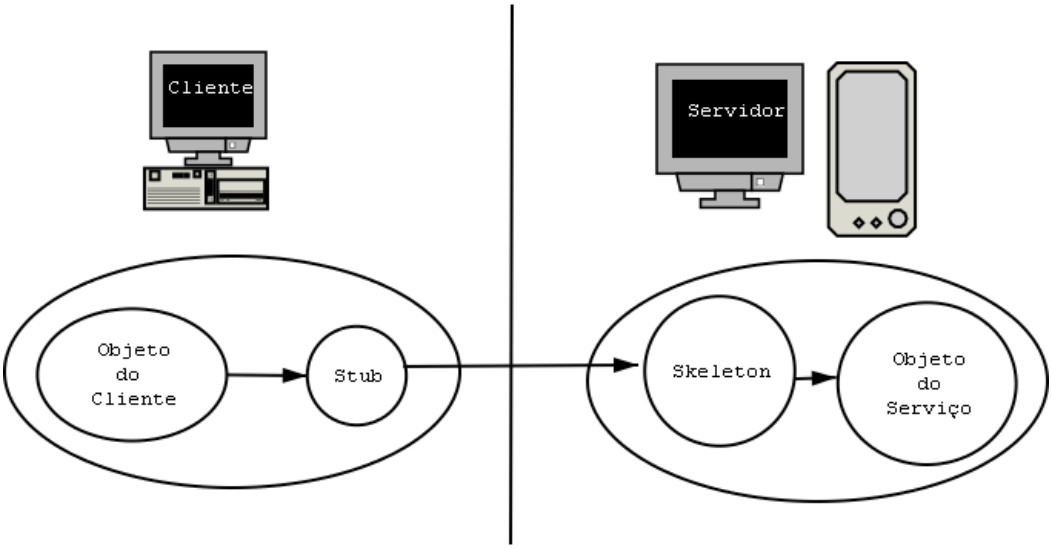

Figura 3.3 Invocação de método com auxiliares. Adaptado de [SIE 2005]

Outro aspecto importante no RMI é a interface remota. O cliente não conhece o objeto com que ele está interagindo. Ele conhece uma interface que será implementada no servidor. Para essa interface dá-se o nome de interface remota. Essa interface permite que o cliente conheça as funções pré-estabelecidas que ele tem acesso no servidor e define essas funções junto com seus parâmetros e retornos. Caso ocorra algo de errado na comunicação, é lançada uma exceção do tipo *RemoteException* que deve sempre ser capturada e tratada pelo cliente. Tanto o Stub, do lado do cliente, quanto o Skeleton, do lado do servidor, são criados a partir dessa interface remota.

No lado do servidor existe uma ou mais classes que implementam realmente essa interface remota. Para que o cliente decida para qual implementação da interface remota deseja se comunicar é necessário que o servidor registre esse objeto em seu serviço remoto. Isso é feito através de um cadastro no registro RMI.

O registro RMI(rmi registry) é similar a uma página de cadastro telefônico, ela possui um nome para o serviço e o local de sua implementação no servidor. Esse registro RMI deve ser sempre iniciado pelo servidor e chamado pelo cliente quando deseja utilizar um dos métodos da interface remota.

Portanto, para criar uma invocação de método remoto utilizando RMI são necessárias cinco etapas[SIE 2005]:

- Criar uma interface remota;
- Criar uma implementação para a interface remota;
- Gerar os stubs e skeletons;
- Iniciar o registro RMI e
- Iniciar o serviço Remoto.

3.2.2 Criando uma interface remota

A interface remota define os métodos que o cliente poderá chamar. A criação de uma interface remota é simples: bastando estender a interface java.rmi.Remote. Assim como java.lang.Serializable, Remote é uma interface que serve apenas de marcação - ela não define métodos a serem implementados e tem um significado apenas para o RMI.

Todos os métodos criados na interface que estende Remote devem lançar uma *RemoteException*. Essa exceção poderá ser lançada caso alguns dos procedimentos de acesso à rede falhem, como por exemplo se a conexão de rede não estiver presente.

Outro detalhe importante é com relação aos retornos dos métodos. Como comentado anteriormente, esses retornos podem ser qualquer tipo primitivo Java, como int, boolean ou char, ou qual objeto que implemente a interface Serializable. Antes de enviar uma mensagem pela rede é necessário que os dados dos objetos sejam serializados e para que um objeto em Java seja serializado ele precisa implementar a interface Serializable.

Serializar um objeto significa transforma os seus dados em bits com uma formatação especial.

3.2.3 Criando uma implementação remota

O cliente deseja chamar um método no servidor remoto. Através da interface remota implementada pelo servidor ele tem acesso aos métodos presentes no servidor. Portanto, o cliente não possui na sua JVM a implementação dos métodos definidos na interface que estendeu Remote. Esses métodos estão no lado do servidor, e para que esses métodos possuam uma implementação real é necessário implementar a interface definida anteriormente. Em Java isso é feito criando uma classe que implemente a interface. Implementar uma interface significa programar os métodos que foram definidos pela interface. Se a interface não definir nenhum método, como Remote e Serializable, ela é chamada de interface de marcação.

Alguns cuidados devem ser tomados nessa etapa: além de implementar a interface, é necessário definir um construtor para a classe que não pode conter nenhum argumento e deve declarar uma *RemoteException*.

3.2.4 Gerando os Stubs e Skeletons

Até a versão 1.4 de Java era necessário, através de um programa chamado rmic (rmi compiler)[4], gerar os stubs e skeletons explicados anteriormente. Isso era feito através de linha de comando. Com o uso de Reflection em Java, a partir da versão 1.5, isso não é mais necessário.

Agora stubs e skeleton são gerados em tempo de execução. Em [MEL 2007] encontra-se detalhadamente o processo a ser realizado.

3.2.5 Executando o rmi registry

O rmiRegistry pode ser iniciado de duas maneiras: via código ou linha de comando. Assim como o rmic, o rmiregistry pode vem junto com o kit de desenvolvimento Java.

Para iniciar o rmiRegistry através do código consultar [MEL 2007].

[4] O rmic vem junto com o kit de ferramenta do Java, até a versão atual (1.6) se encontra sob o pacote bin. Para gerar os stubs e skeletons basta rocar o rmic na classe de implementação do serviço.

3.2.6 Iniciar o serviço Remoto

Iniciar o serviço Remoto significa instanciar um objeto da classe que contem a implementação remota e publicar esse serviço no rmiRegistry para que os clientes tenham acesso. Mais detalhes em [MEL 2007].

3.2.7 Avaliação de RMI

A principal vantagem do RMI é de já ser um "pacote" incluso na plataforma Java. Não é necessário adicionar nenhum jar extra ou configurar qualquer documento extra. Por ser uma tecnologia Java, vantagens como portabilidade, uso de Orientação a Objetos e Padrões de Projeto são herdadas automaticamente.

A documentação on-line é extensa. A SUN oferece mini-cursos e tutoriais. Além disso, a maioria dos livros escritos sobre Java falam de RMI. Como está presente desde a versão 1.1 do Java, RMI amadureceu junto com a plataforma e não é mais necessário criar stubs ou invocar o rmiregistry via linha de comando. Isso tudo devido a tecnologias inclusas na linguagem como, por exemplo, Reflection.

Apesar disso tudo, em comparação com o framework tratado a seguir o DualRPC o RMI ainda é um pacote de muito baixo nível. Não oferece suporte a chamadas assíncronas e o tratamento de threads é feito somente no lado do servidor. Além disso, não é um recurso fácil de usar: necessita-se de uma série de técnicas para fazer coisas simples, como enviar mensagens do servidor para o cliente ou para reconhecer o cliente que efetuou a chamada. Além disso, RMI não estabelece uma sessão entre cliente o servidor. O cliente executa chamadas em classes que estão em um servidor e nada além disso.

Outro problema que RMI possui é que não é possível trabalhar com clientes em outras linguagens, isto é, somente Java é suportado.

Para o trabalho em questão, esses requisitos são extremamente importantes. No início do projeto a solução adotada foi o RMI. Conforme o andamento do projeto, notou-se que o RMI não suportava alguns requisitos, como por exemplo: realização de chamadas assíncronas, túnel entre cliente e servidor e chamadas do servidor no cliente. Seria necessário implementar essa série de requisitos "a mão", o que ia demandar em um tempo

extra para isso e um estudo mais aprofundado. Buscou-se uma alternativa, quando o DualRPC apareceu como uma ótima opção.

3.3 DualRPC

DualRPC é um framework Java para realizar chamada de procedimentos remotos entre cliente e servidor. O framework possui a característica de ser bidirecional, um cliente pode chamar um método no servidor e o servidor pode chamar métodos no cliente. Isso acontece porque é criada uma conexão TCP/IP contínua, e o estado entre cliente e servidor é preservado entre as chamas RPC.

A documentação de DualRPC e o download do artefato cliente e servidor pode ser encontrada em: http://www.retrogui.com.

DualRPC foi projetado para ajudar desenvolvedores a implementarem aplicações distribuídas de forma rápida. Construir aplicações em DualRPC é simples quando comparado com RMI, por exemplo. O desenvolvedor basicamente precisa criar duas classes para comunicação: uma do lado do cliente e outra do lado do servidor. Essas "classes de comunicação" contêm métodos que serão expostos para chamadas RPC.

Internamente, o framework usa conexão via sockets e cria um canal contínuo via TCP/IP. Quando o cliente se conecta no servidor, um par de objetos servidores é criado, um no cliente e outro no servidor. É dessa forma que é possível para o servidor também chamar o cliente.

Assim como RMI, o DualRPC suporta qualquer tipo Java ou classes (desde que sejam serializáveis). Diferente do RMI, não é necessário gerar qualquer stub ou skeleton para que ocorra a comunicação. O interessante no dualRPC é que clientes podem ser implementados em plataformas diferentes de Java usando de protocolos como SOAP, XML_RPC, JSON e outras.

3.3.1 Funcionamento

DualRPC utiliza sockets TCP/IP para realizar as trocas de mensagens. Com DualRPC uma sessão entre cliente e servidor é criada do início até o fim da conexão Isso significa que o servidor sabe exatamente qual cliente está realizando a chamada.

As mensagens trocadas contêm informações a respeito do método que deve ser chamado, como por exemplo, a classe em que se encontra o método, seus parâmetros, etc.

As duas principais classes fornecidas pelo framework são DualRpcServer e DualRpcClient.

Para criar um servidor basta instanciar um objeto da classe DualRpcServer Na instanciação é necessário informar apenas o host e a porta na qual o servidor deseja ficar "escutando". Algumas configurações podem ser realizadas, como escolher um número máximo de conexões, utilizar JAAS, etc. Basta então chamar o método "listen()" do DualPpcServer e o servidor estará no ar.

Para criar um cliente basta instanciar um objeto da classe DualRpcClient. Esse objeto também é instanciado passando o endereço do servidor e para realizar a conexão basta chamar o método connect().

Esses objetos conectados indicam que existe uma sessão entre cliente e servidor e um conhece o outro. No servidor o cliente ganha um clientSessionId, um identificador único do cliente no servidor.

A partir desse momento é chamar procedimentos tanto por parte do cliente quanto por parte do servidor.

Os procedimentos no DualRPC podem ser classificados em dois tipos: síncronos e assíncronos.

Nos procedimentos síncronos, o lado que chama o procedimento fica esperando uma resposta do outro lado, assim como uma o procedimento chamado localmente. Somente quando o procedimento é executado a resposta é retornada e o programa pode continuar executando. Nessas mensagens é possível que haja um tipo de retorno. Esse tipo de retorno no DualRPC não pode ser nulo. Se ocorrer algum problema e o servidor não conseguir realizar a chamada limite de em um tempo limite (timeout), uma CallException é lançada para o cliente.

Os procedimentos assíncronos, por outro lado, não exigem que o lado que realizou a chamada fique “esperando”. Nesse tipo de procedimento existe a certeza que o mesmo foi chamado, caso contrário uma CallException seria lançada, porém não existe a certeza de que o procedimento foi realizado sem erros. Obviamente, esses procedimentos não devem ter tipo de retorno, o que significa que todos devem ser declarados como *void* em Java.

Para chamar um procedimento remoto deve-se invocar os métodos “call()” e “callAsync()” para chamadas síncronas e assíncronas respectivamente, nos objetos instanciados de DualRpcServer e DualRpcClient.

Uma chamada de procedimento sempre apresenta pelo menos três parâmetros: o método a ser chamado do outro lado (um String), um ou mais parâmetros para o método que será chamado e uma classe tratadora do método.

3.3.2 Classes Tratadoras

As classes tratadoras (também chamadas de Handlers) são semelhantes aos objetos que implementam as interfaces remotas no RMI. São objetos dessas classes que recebem e tratam os métodos chamados pelos clientes.

Quando um cliente chama um método no servidor é necessário que ele especifique qual a classe tratadora que possui esse método no servidor. Assim que a mensagem chega, um novo tratador é instanciado para o cliente e mantido até que o cliente se desconecte. Os métodos que vem em seguida usam portanto o mesmo tratador.

Uma classe tratadora estende uma das classes abstratas: AbstractClientRpcHandler, para tratadores no lado do cliente, ou AbstractServerRpcHandler para tratadores no lado do servidor.

É necessário, do lado do servidor, cadastrar todas as classes que podem ser chamadas pelo cliente, para isso usa-se o método registerServerSideHandlerClassname(String handlerClassname) no objeto da classe DualRpcServer. Como um servidor pode ter mais de um tratador e esses tratadores são instanciados pelo servidor e precisam ser registrados antes de serem chamados pelos clientes.

No lado do cliente existe a mesma necessidade, porém normalmente usa-se apenas um tratador, que é registrado através do método

registerClientSideHandler(AbstractClientRpcHandler clientRpcHandler) que recebe o objeto já instanciado.

3.3.3 Avaliação do DualRPC

Como já mencionado, a principal vantagem de DualRPC sobre RMI é a facilidade de desenvolvimento. DualRPC demanda muito menos esforço para codificar, depurar e manter uma aplicação. O processo para fazer RMI funcionar envolve colocar o rmi registry no ar, gerar stubs e prover o carregamento dinâmico de código. Isso torna muito mais complexa a criação de aplicações, mesmo as simples. Isso pôde ser verificado, já que no início do projeto tentou-se utilizar. RMI. Os problemas ocorridos com DualRPC foram muito mais facilmente solucionados,, mesmo com documentação mais escassa.

DualRPC não é um framework que gera impactos no código ou na estrutura da aplicação, como pôde ser visto durante o trabalho.

DualRPC também provê duas formas de realizar chamadas remotas com um único socket, que são chamada Síncrona e Assíncrona (tratadas anteriormente).

Outra característica de DualRPC não usada nesse trabalho mas que é muito interessante, é que possui SSL, JASS, login, HTTP e SOCKS proxy, o que possibilita criar aplicações mais seguras.

A principal desvantagem de DualRPC em relação a RMI, é que foi idealizado como um framework para transferir parâmetros pequenos em uma chamada de procedimento. Isso significa que os parâmetros de um procedimento que é chamado devem ser minimizados[5] para que isso não prejudique a eficiência da comunicação. Além disso, segundo a documentação, um único servidor pode suportar teoricamente até 1500 (mil e quinhentas) conexões ativas ao mesmo tempo.

Quanto ao fato de transferir pequenos parâmetros em uma chamada de procedimento, o autor não é muito claro. No trabalho realizado, apesar de o cliente poder escolher o tamanho das mensagens que serão enviadas, isso não é um grande problema: basta que o cliente envie apenas as informações necessárias para realizar uma jogada. Por maior que seja o jogo, as informações referentes a jogadas normalmente dizem respeito

[5] Por minimizados, entenda-se que a quantidade de bytes transferida não deve ser demasiadamente grande, de preferência não ultrapassar a casa dos Kbytes..

apenas a uma mudança de estado. Nos projetos desse trabalho onde foi implantado o framework, esses dados não passavam de alguns atributos em um único objeto.

Quanto à segunda desvantagem, a limitação da quantidade de conexões, produz impacto, originando uma deficiência do servidor produzido. Na conclusão apresenta-se como um dos trabalhos futuros tornar o servidor mais escalável, conforme a demanda necessitar.

4. Os artefatos de software produzidos

Em última análise, o que o desenvolvedor de um jogo deseja é uma forma de enviar e receber mensagens pela rede. Essas mensagens possuem um significado muito específico para o jogo tratado. Para um jogo de cartas, essas mensagens podem significar as cartas que um adversário acabou de descartar e que modificaram o estado do jogo. Já para um jogo de tabuleiro, essas mensagens podem significar o número que saiu no lançamento dos dados do adversário ou o número de casas que ele deve movimentar seu peão.

Após estudo sobre a gama de jogos que é passível de desenvolvimento (ver capítulo 2.1.1), foi planejado um protocolo para essa troca de mensagens que pudesse ser o mais flexível possível e que atendesse aos requisitos estabelecidos anteriormente.

Com base nesse protocolo foram implementados:

- Um aplicativo servidor, que tem por função gerenciar os programas que desejem realizar partidas, incluindo recepção e envio de mensagens aos clientes/jogadores de um determinado jogo e
- Um framework que permite acesso ao servidor e que facilita a construção de aplicativos de jogos por parte dos desenvolvedores.

Pelos motivos citados no capítulo 3, foi escolhido o framework DualRPC para realizar as trocas de mensagem. Dessa forma, o trabalho de desenvolvimento consistiu na construção do sistema servidor e do framework, em linguagem Java.

4.1 O protocolo

A partir dos casos de uso[6] foi possível escrever o protocolo de troca de mensagens. O protocolo implementado é bastante simples (ver figura 4.1). Ele é constituído basicamente de três fases: Conexão, Jogada(s)[7] e Desconexão.

[6] Os casos de uso podem ser encontrados em anexo II

[7] Entende-se por jogada, toda e qualquer mensagem trocada entre cliente e servidor na realização de uma partida.

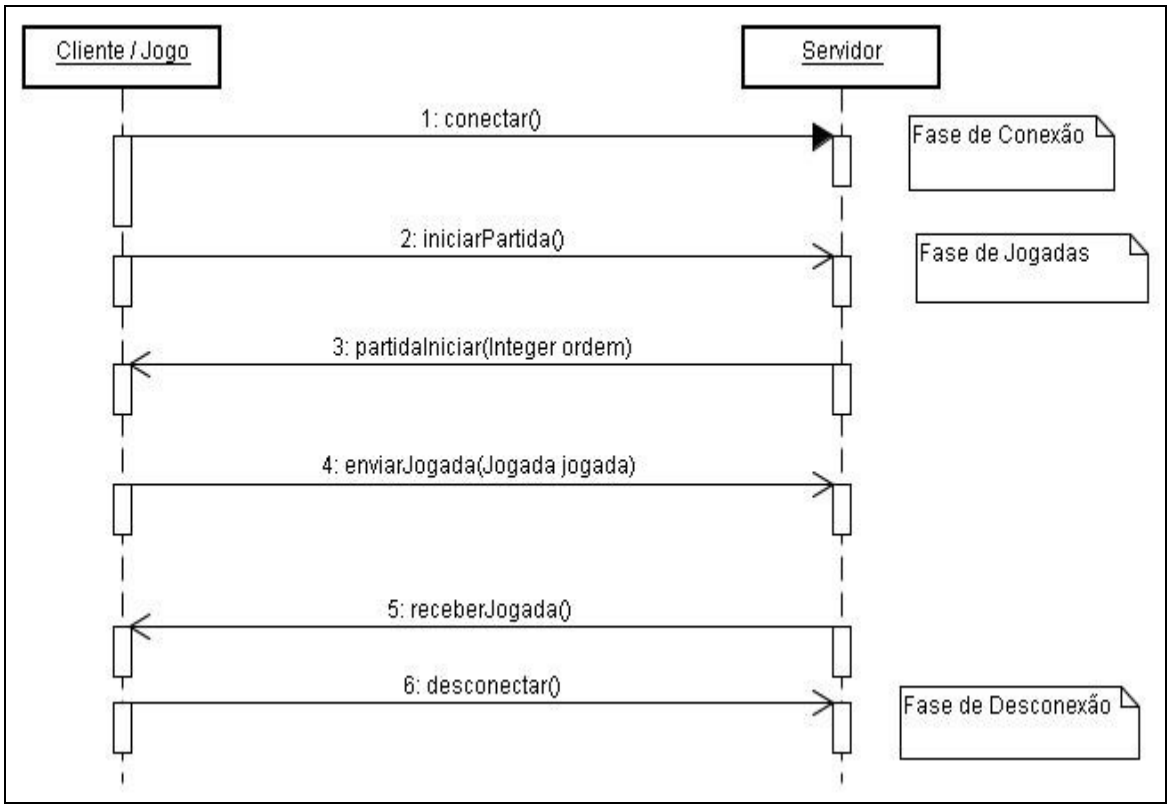

Figura 4.1 Diagrama de seqüência do protocolo.

Diversos aspectos e requisitos não funcionais foram levados em conta na construção do protocolo. Uso de banda foi o principal deles. Quando um usuário se conecta no servidor, ele não recebe uma resposta ou um método é chamado para indicar que ele está conectado. Isso porque se ele não conseguir se conectar uma exceção será automaticamente lançada no lado do cliente.

O primeiro intuito foi construir um protocolo onde somente o cliente acessasse o servidor e chamasse um método "perguntando" se existia alguma informação para ele. Essa abordagem foi logo descarta pelo consumo de banda extra. Mesmo se não houvesse informação para o cliente, ele ficaria sempre fazendo requisições ao servidor.

Outro requisito importante é a assincronicidade. O cliente não deseja chamar um método no servidor e ficar esperando enquanto o método é processado para então obter uma resposta. A maioria dos métodos chamados é assíncrona, isto é, o cliente chama o método e

continua a execução do seu programa. Se um dos métodos resultar em uma ação no jogador que o chamou, essa ação é chamada pelo servidor quando o método chamado for executado - como por exemplo o método partidaIniciada(). O framework dualRpc já possui chamadas de métodos assíncronos.

4.1.1 A fase de conexão

Para que um cliente se conecte ao servidor são necessárias: um nick e o identificador do Jogo no qual o cliente será conectado.

O nick é uma cadeia de caracteres que serve como identificador do jogador no jogo. Esse nick é escolhido pelo usuário na hora de se conectar. Caso essa cadeia seja vazia ou nula, o cliente ganha um nick padrão no sistema: “jogador”.

O identificador do Jogo é também uma cadeia de caracteres que identifica um aplicativo dos demais. Cada jogo desenvolvido tem, portanto, um id único e comum entre todos os jogadores. Esse id é obtido ao cadastrar o aplicativo de jogo desenvolvido no site: http://java.inf.ufsc.br/games.

Quando o desenvolvedor cadastra o jogo, ele recebe um arquivo chamado jogoMultiPlayer.properties. O arquivo deve ser colocado no classPatth da aplicação para que o framework possa reconhecer esse id. Sem esse arquivo, o framework lança uma exceção *ArquivoNaoEncontradoException* quando utilizado.

Após receber uma requisição de um jogador, a primeira ação realizada pelo servidor é verificar se algum aplicativo com o mesmo identificador do jogo já se encontra em execução. Caso o cliente seja o primeiro a se conectar, o servidor cria uma instância do jogo que o cliente requisitou e aloca o cliente no jogo determinado. Todos os clientes vindo após o primeiro são automaticamente enviados para a instância de jogo antes criada.

4.1.2 A fase de troca de Jogadas

Nessa fase do protocolo o cliente já se encontra conectado e pode enviar e receber mensagens do servidor. É possível que o usuário não chegue a realizar essa fase e peça para se desconectar logo após ter se conectado.

Nessa fase podem diversos métodos podem ser chamados pelo cliente no servidor, esses métodos servem para: iniciar partida, enviar e receber jogadas e reiniciar uma partida em andamento ou finalizad**a.**

4.1.2.1 Iniciar partida

Nesse momento o jogador deseja iniciar uma nova partida, ou seja, criar uma sessão formada por ele e outros jogadores, onde o jogo em si possa ser realizado. Ao iniciar uma partida, o jogador envia as informações de quantos jogadores estarão presentes na determinada partida. Dessa forma, o servidor seleciona entre os jogadores livres, o número de jogadores da partida.

Os jogadores selecionados são colocados dentro de uma partida e uma mensagem para iniciar a partida é chamada nesses jogadores, para que eles se preparem para a partida que será iniciada.

Portanto, uma nova partida, do ponto de vista do jogador, pode ser iniciada de duas maneiras, ou ele solicita o inicio de uma partida, ou é selecionado pelo servidor para participar de uma partida. Esse modelo dispensa o uso de convites, onde o servidor enviaria uma mensagem para cada jogador perguntando se ele desejaria entrar em uma partida. Essa solução foi adotada por simplificar o modelo e o protocolo.

Caso o jogador seja selecionado para uma partida, mas não deseje continuar, basta iniciar uma partida como solicitante. A partida em que ele se encontrava será automaticamente fechada e os outros jogadores receberão uma mensagem de que um dos jogadores desistiu.

Através dessa da referência para a partida, o jogador poderá efetuar suas jogadas e será acessado para que as jogadas dos outros jogadores sejam atualizadas no seu aplicativo jogo.

4.1.2.2 Enviar e Receber jogadas

O envio e recepção de jogadas em uma partida é realizada de forma assíncrona. Ao enviar uma jogada, deve-se tratar algumas exceções. Se nenhuma delas for lançada, então a mensagem foi garantidamente enviada ao servidor. Ao tratar a jogada, o servidor a

envia a todos os usuários que estão participando da mesma partida, que recebem a mensagem. A recepção de mensagens é feita pela classe tratadora do cliente (presente no framework) e enviada para os ouvidores presentes na aplicação. A partir daí, o trabalho é com o desenvolvedor. Ele deve definir a melhor forma de tratar essas mensagens.

4.1.2.3 Reiniciar partida

Como o servidor não tem controle sobre as partidas que estão acontecendo, não é possível saber se uma partida foi finalizada. A partir do momento que uma partida é iniciada no servidor ela só termina quando o método *finalizarPartida()* é chamado ou quando um dos jogadores se retira da partida.

No lado do jogador, uma partida termina sempre que a vitória de um dos jogadores é declarada ou ocorre um empate. Após isso, diversas decisões podem ser tomadas pelo lado dos jogadores, uma delas é iniciar uma nova partida com outros jogadores, nesse caso o método "*iniciarPartida()"* explicado anteriormente é usado.

Caso o jogador deseje iniciar uma partida com os mesmos jogadores, basta que ele chame o método reiniciarPartida(). O método apenas envia uma mensagem a todos os jogadores presentes na partida de que devem se preparar para o novo inicio de um jogo.

Esse método também pode ser chamado durante uma partida, cabendo ao desenvolvedor habilitar ou desabilitar essa opção.

4.1.3 A fase de desconexão

O protocolo de desconexão é bastante simples. O cliente que pede para ser desconectado é retirado do jogo e das partidas onde se encontrava. Caso o jogador esteja em alguma partida, além de ser retirado da mesma, uma mensagem é enviada a todos os outros jogadores, já que em casos como este não é possível continuar um jogo. Todos os outros jogadores são automaticamente retirados da partida e é enviada uma mensagem de fim de "partida".

Essa fase do protocolo também pode acontecer caso o usuário, por algum motivo, for desconectado do jogo, como, por exemplo, se a conexão for perdida.

5. O servidor multi-jogos

O principal requisito do servidor de jogos é que ele deve ser genérico o suficiente para suportar o desenvolvimento de diversos jogos sem exigência de tempo real. Outros requisitos importantes dizem respeito a escalabilidade: o servidor deve suportar ao mesmo tempo mais de um jogo, e garantir que cada jogo possa ter diversos jogadores conectados participando em diversas partidas.

O diagrama de classes do projeto construído a partir desses requisitos leva em conta também o framework escolhido para realizar todas as comunicações, o dualRPC.

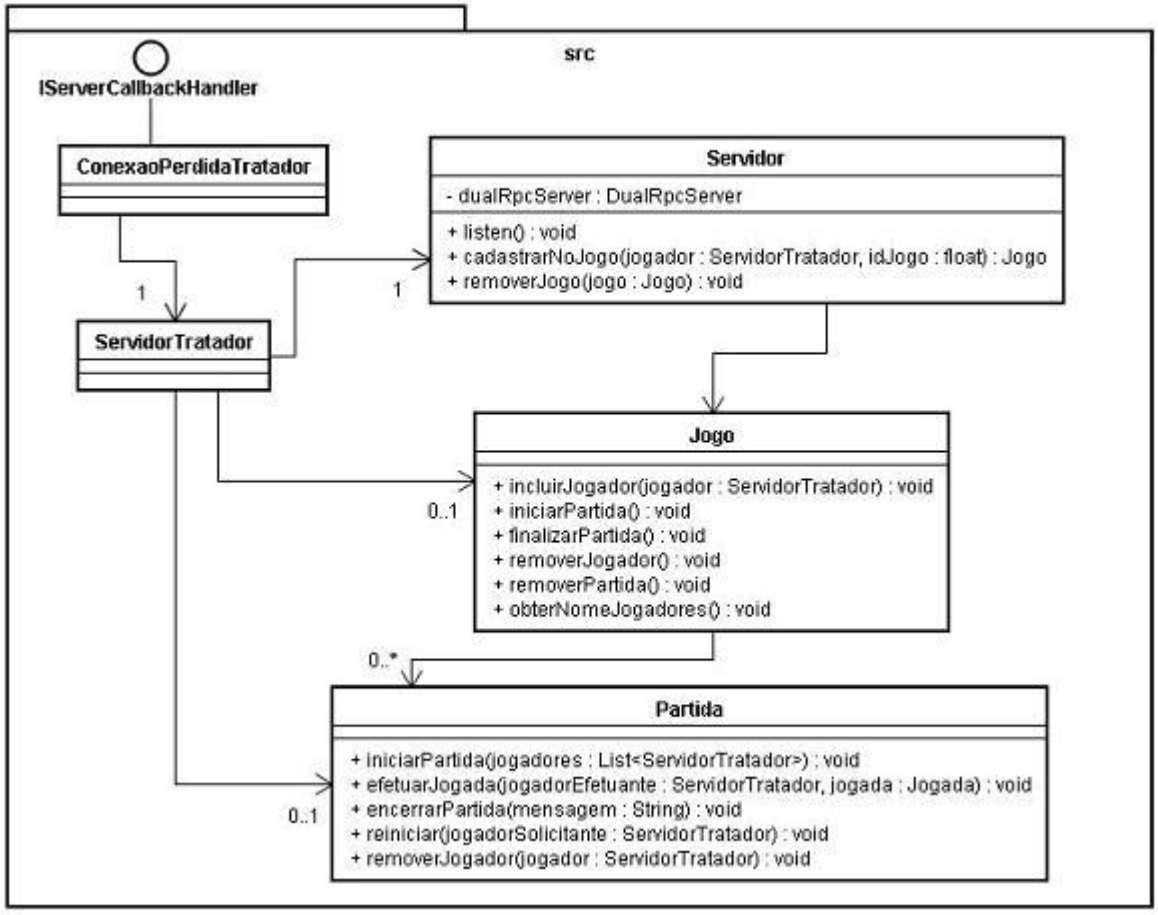

Figura 5.1 Diagrama de classes do servidor

Como pode ser visto na figura 4.2[8], o servidor possui uma estrutura bastante simples. Existe um *Servidor* central onde os jogadores inicialmente se conectam, e são

[8] Esse diagrama de classes está simplificado, ele apresenta apenas as classes essenciais

direcionados para um determinado *Jogo* no servidor. Esse Jogo representa uma aplicação do cliente. Para um jogo da velha existe uma instância de jogo no servidor, para um jogo de xadrez existe outra, e assim para cada jogo diferente dos anteriores. Dentro de um jogo onde os clientes estão conectados é possível realizar partidas. Uma *Partida* representa uma sessão onde dois ou mais jogadores efetuam e recebem jogadas.

Quando uma conexão com DualRpc é efetuada, é criada uma sessão entre cliente e servidor. O representante do cliente no lado do servidor é chamado de *ServidorTratador.* Todos os métodos chamados pelo cliente são direcionados para o seu ServidorTratador no lado do servidor.

Caso a conexão entre cliente e servidor seja perdida, um objeto de *ConexaoPerdidaTratador* é chamado e comunica ao ServidorTratador que a conexão foi perdida.

5.1 Funcionamento

O servidor trabalha em cima do protocolo explicado anteriormente. para cada fase do protocolo existem métodos que são chamados no servidor. O servidor trata esses métodos, executa ações e chama métodos no cliente, conforme os métodos são tratados..

O servidor se comunica com o cliente através do framework tratado na seção 4.3.

5.2 Conexão

Quando um cliente[9] requisita uma conexão com o servidor através do dualRpcClient, um ServidorTratador é criado no servidor. Esse ServidorTratador faz parte de uma sessão e o cliente recebe um clientSessionId que identificará esse cliente dos demais, no servidor.

Ao se conectar, o cliente informa também qual o aplicativo que ele está executando. Isso é informado através de um identificador próprio, localizado pelo framework e que faz

[9] Para o servidor, o cliente é sempre o framework desenvolvido. Portanto, todas as ações realizadas pelo cliente no servidor são realizadas através do framework, assim como as ações do servidor são encaminhadas para o framework que propaga isso para a aplicação cliente.

parte de um arquivo chamado “jogoMultiPlayer.properties”, que deve ser colocado no classpath da aplicação.

Logo, quando um cliente se conecta, uma instância de Jogo é criada no servidor. Os clientes que se conectarem e tiverem um identificador igual são também enviados ao mesmo jogo.

O processo de conexão de um cliente no servidor envolve, portanto, a criação de uma instância de Jogo que será compartilhada por todos os aplicativos clientes que estão rodando o mesmo aplicativo de jogo Se o jogo já foi instanciado e possui alguém conectado, são encaminhados a ele os próximos clientes que pedirem conexão para o mesmo aplicativo de jogo.

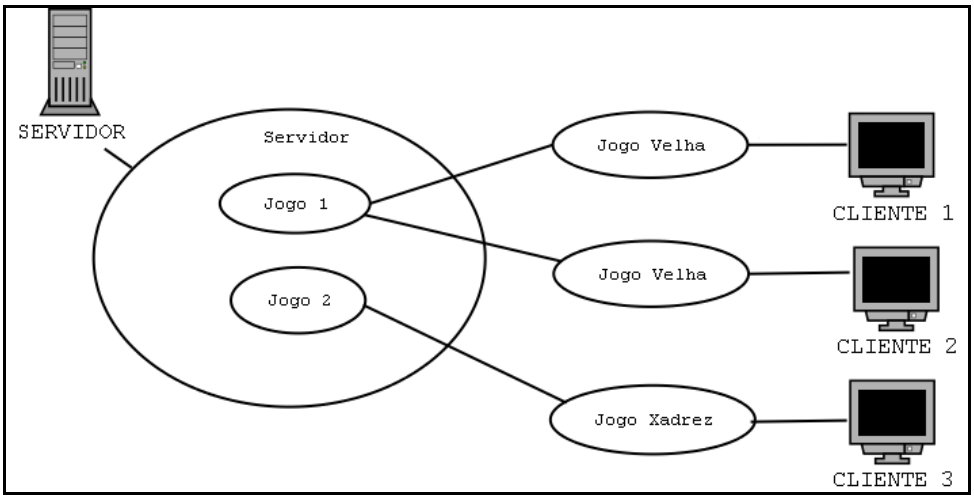

Figura 5.2 Exemplo de aplicativos conectados no servidor.

Como pode ser visto na figura 4.3[10], em um determinado instante do tempo o servidor pode estar com 3 clientes conectados. Os clientes 1 e 2 estão rodando o mesmo jogo (Jogo da Velha). Portanto, no servidor existe apenas uma instancia que é compartilhada por ambos. Já o cliente 3, está rodando um outro jogo (Jogo de Xadrez). Quando o cliente 3 pediu a conexão no servidor, o servidor criou o jogo 2 porque o identificador do jogo passado pelo cliente 3 era diferente do identificador do jogo da velha.

[10] Apenas por simplificação os objetos ServidorTratador foram desconsiderados, porém para cada conexão de um cliente com o Jogo existe um ServidorTratador que invoca os métodos de Jogo.

5.3 Jogadas

As jogadas entre os clientes de um mesmo jogo são realizadas em partidas. Uma partida possui um número fixo de jogadores que trocarão mensagens, a fim de atingir um objetivo em comum. As mensagens são enviadas por um dos jogadores e recebidas automaticamente por todos os demais

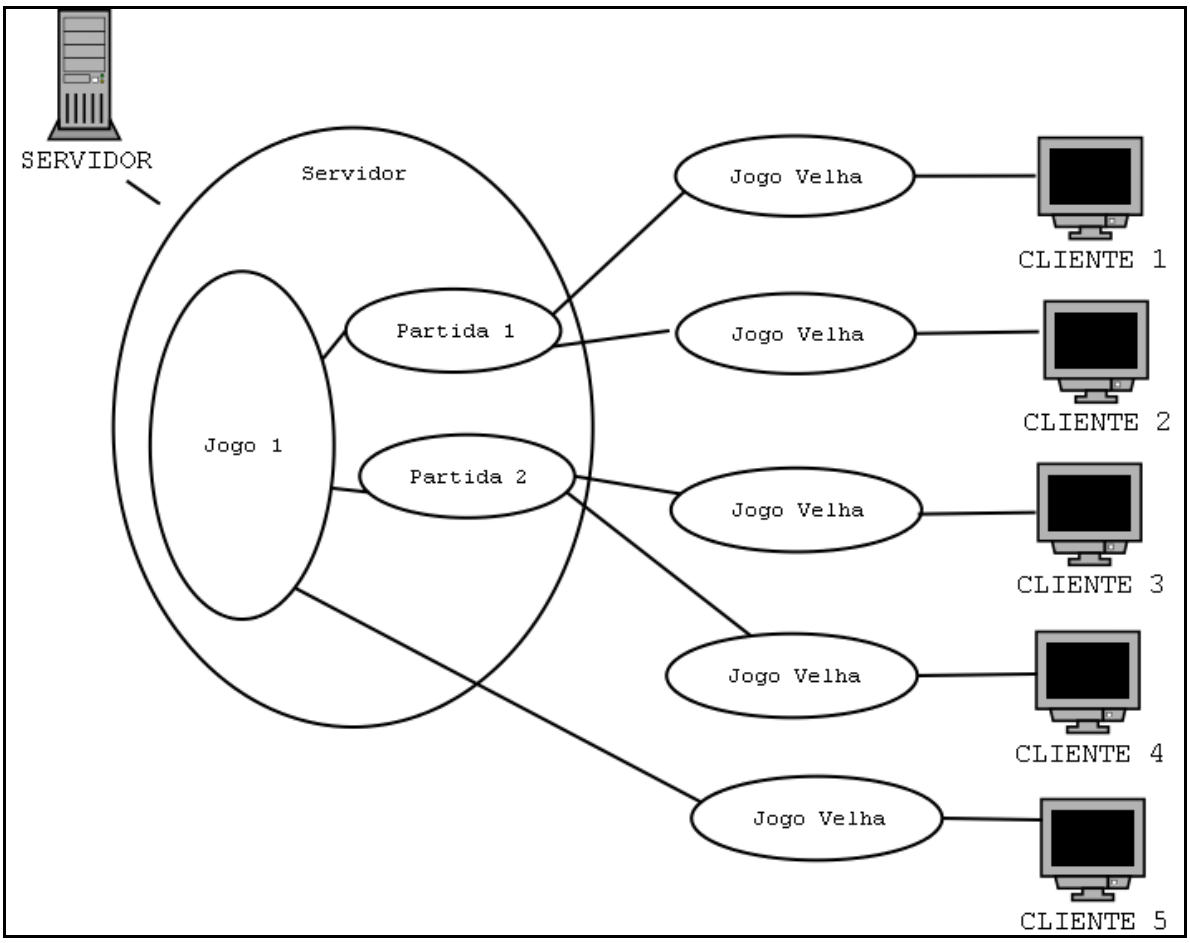

Figura 5.3 Exemplo de aplicativos realizando partidas no servidor

Como pode ser observado na figura 4.4, em um determinado instante do tempo, podem existir diversas partidas em uma instância de Jogo, cada cadastrada no jogo e possuindo um número fixo de jogadores.

5.4 Desconexão

Quando um cliente pede a desconexão no servidor, ele chama o método desconectar do seu ServidorTratador associado. Esse método indica para o servidor que o cliente associado se desconectou e a conexão foi perdida.

O servidor recebe essa mensagem e a encaminha para o jogo onde o cliente estava conectado. O jogo exclui o jogador de qualquer partida que ele esteja participando.

5.5 Ordenação dos jogadores

Uma das características que os jogos implementados pelo ambiente podem ter que tratar é a ordenação dos jogadores. A principio, a ordenação dos jogadores é definida no servidor. Quando um jogador faz a requisição para iniciar uma partida, o servidor automaticamente escolhe seus adversários e também define uma ordem de jogadas, sendo que o jogador que solicitou a partida é sempre o primeiro a jogar.

Existem, porém, situações onde os jogadores desejam mudar essa ordenação - seja através de escolha ou sorteio. Por isso, o framework disponibiliza os métodos getListaOrdenadaJogadores() e setListaOrdenadaJogadores().

Caso o usuário queira montar essa ordenação, o primeiro passo consiste em requisitar do servidor a lista de jogadores, como dito anteriormente, basicamente o que diferencia um jogador é seu identificador. Logo, o que o desenvolvedor recebe como ferramenta de trabalho é uma lista com os "nomes" dos jogadores. A partir daí, o desenvolvedor atua da forma que considerar melhor, desde que ao final de sua iteração sobre a lista, o resultado seja uma outra lista, com os mesmos elementos, ordenada conforme sua vontade. A lista é validada através do Proxy, que então envia a nova requisição de ordenação ao servidor. O servidor encaminha essa ordenação a todos os outros jogadores.

6. O framework

O framework[11] construído tem como objetivo principal facilitar a vida do desenvolvedor de jogos, no que diz respeito ao acesso ao servidor. Caso nada fosse feito, o desenvolvedor necessitaria aprender sobre o framework DualRpc, seria necessário disponibilizar a interface utilizada pelo servidor para que o desenvolvedor tivesse acesso e a partir daí, o acesso se tornaria imprevisível.

Além disso, existem diversas validações que seriam mais eficientes se fossem feitas no lado do cliente, antes de enviar a mensagem para o servidor, como no exemplo onde um usuário chama o método enviarJogada sem iniciar uma partida previamente. Essa mensagem é aceita pelo servidor, mas como o cliente não está jogando uma partida com outras pessoas ele gera uma exceção. Para esse caso, é muito mais interessante que a exceção seja lançada na parte do cliente assim que o envio da mensagem for requisitado.

Devido a características do DualRpc, é possível utilizar outras plataformas e linguagens de programação como por exemplo C/C++ e criar clientes que se comuniquem com o servidor utilizando protocolos como XML-RPC, SOAP ou Json.

Os requisitos para a construção do framework, portanto, são essencialmente:

- Garantir o acesso ao servidor de forma transparente para o usuário.
- Validar os métodos de acesso ao servidor no lado do cliente.
- Ser genérico o suficiente para ser usado nos jogos sem exigência de tempo real.
- Montar uma interface para acesso ao servidor.

A arquitetura do framework foi construída conforme o diagrama de classes da figura a 4.5:

[11] Maiores detalhes sobre frameworks podem ser encontrados em [WIR 1990], [WIR 1197] e [SIL 2000]

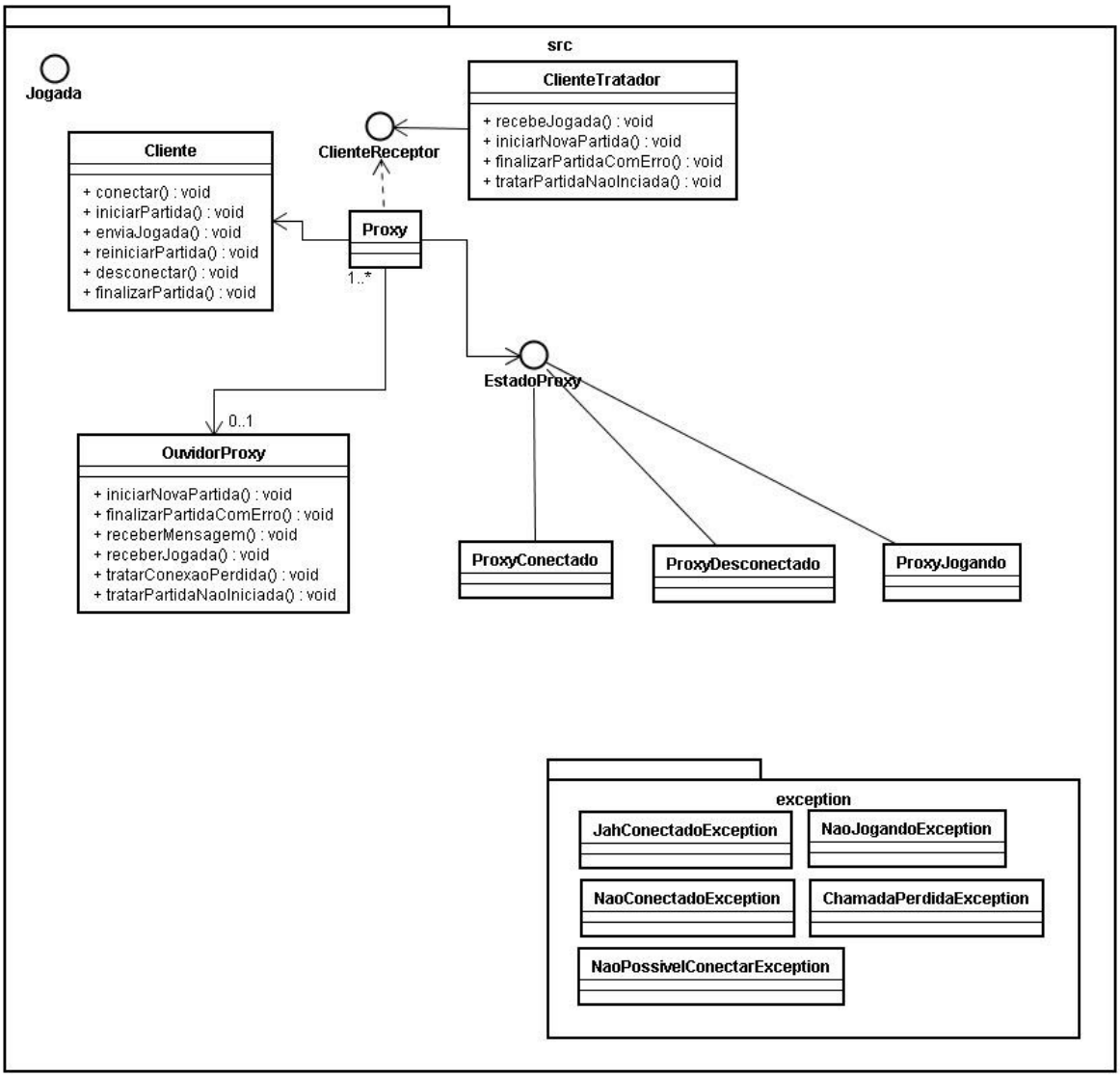

Figura 6.1 Diagrama de classes do framework

Como é possível observar na figura,[12] a arquitetura construída é bastante simples. O desenvolvedor de jogos deve se preocupar com apenas dois pontos essenciais ao utilizar o framework:

- Como enviar mensagens para o servidor;
- Como receber as notificações do servidor.

[12] Esse diagrama de classes está simplificado, ele apresenta apenas as principais classes.

A forma de enviar as mensagens ao servidor é bastante simples. Como pode ser visto no diagrama acima, foi definida uma classe chamada *Proxy*. Essa classe possui todos os métodos de acesso ao servidor por parte do framework.

A classe Proxy possui duas características básicas. A primeira é que ela só é instanciada uma vez durante toda a execução do programa. Ela implementa o padrão de projeto Singleton [GAM 1994]. Portanto, para capturar a instancia dessa classe, basta chamar o método *getInstance( )*.

Os motivos de ela implementar o padrão singleton provêm dos requisitos do sistema: cada cliente só pode se conectar ao servidor uma vez. Se houvesse mais de um Proxy no programa, o desenvolvedor poderia facilmente se conectar duas vezes ao servidor. Também seria possível, através do uso de Thread's em Java, invocar dois métodos no servidor de modo à invalidar os jogos.

A segunda característica é o uso do padrão de projeto Observer e envolve os objetos que receberão as mensagens provindas do servidor. Essas mensagens são as enviadas pelas outras instâncias da mesma aplicação que estão, de alguma forma, modificando os seus estados. Nesse caso, foi utilizado o padrão de projeto Observer, onde objetos que implementem a interface OuvidorProxy (fornecido no framework) são registrados no proxy, e assim que o servidor chama o proxy para transferir uma mensagem, esses ouvintes são automaticamente notificados.

6.1 Funcionamento

É descrita a seguir a forma como uma mensagem parte do framework para o servidor e como o servidor chama as mensagens no cliente através do framework[13]. Baseado no padrão no Proxy [GAM 1994], foi definida uma classe de comunicação com o servidor. Essa classe mascara do cliente quaisquer detalhes da comunicação.

A classe Proxy é a implementação da interface servidor. Ela possui os métodos que poderão ser chamados no servidor pelo cliente. Outra função da classe proxy é gerenciar os ouvidores. Os ouvidores serão objetos do lado do cliente que receberão mensagens do servidor a qualquer momento da comunicação.

[13] Em Anexo I se encontram os procedimentos de como adaptar um jogo stand alone sem exigência tempo real para o framework desenvolvido

De acordo com o que foi explicado no protocolo, existem três etapas bem definidas de comunicação: Conexão, Jogando e Desconexão. Essas etapas refletem no Proxy e nas validações executadas por ele. Um cliente só pode enviar e receber jogadas se ele estiver efetivamente jogando, ele também pode requisitar uma desconexão após estar conectado. Baseado nessas validações foi implementado o padrão de projeto State [GAM 1994].

O estado inicial do cliente é DESCONECTADO, indicando que nenhuma mensagem pode ser trocada entre servidor e cliente nesse momento. Para que o cliente comece a trocar mensagens é necessário que ele se conecte ao servidor. A partir desse momento o estado do proxy é CONECTADO. Nesse estado o cliente e servidor já possuem uma conexão e já está criada uma sessão para o cliente no lado do servidor. A partir desse momento, para que possa jogar e se conectar a outros clientes através do servidor, é necessário que o jogador deseje iniciar uma nova partida ou que seja selecionado para uma nova partida. O estado do Proxy passa a ser JOGANDO. Nesse estado o cliente pode enviar ou receber jogadas. Isso é feito de forma sincronizada para que um jogador não receba e envie mensagens ao mesmo tempo. A partir desse estado, o jogador pode voltar a ser CONECTADO ou então pedir a desconexão do servidor. Se o jogador requisita a desconexão com o servidor ele volta ao seu estado original, ou seja, DESCONECTADO.

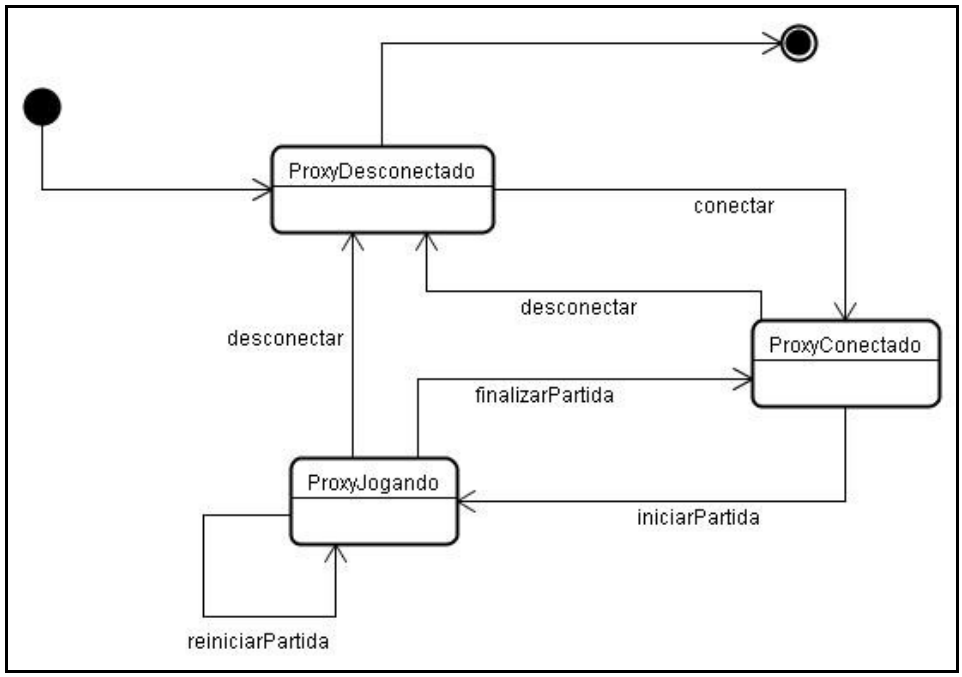

Figura 6.2 Diagrama de estados da Classe Proxy

Quando um desenvolvedor captura a instância da classe Proxy, ele pode chamar qualquer um de seus métodos públicos. Porém, existem ações que só podem ser tomadas quando o Proxy se encontra em um determinado estado. Essas ações foram agrupadas como segue:

Proxy Desconectado:

- Conectar

Proxy Conectado:

- Desconectar
- Iniciar uma partida

Proxy Jogando:

- Desconectar
- Iniciar Partida
- Reiniciar Partida
- Enviar Jogada
- Finalizar Partida

Quando uma ação que não corresponde a um estado é chamada, ela lança uma exceção. Essa exceção deve ser capturada e tratada pelo desenvolvedor. Essas exceções são exceções de validação. Elas validam se no estado em que o Proxy se encontra é possível chamar o determinado método. São elas:

- NaoJogandoException: Lançada caso o cliente tente chamar um método que exige que ele esteja jogando uma partida com outro(s) cliente(s). Como por exemplo, enviarJogada();
- NaoConectadoException: Lançada caso o usário tente chamar um método que exige que ele esteja conectado, como, por exemplo, inicarPartida() ou desconectar();
- JahConectadoExcetion: Lançada caso o usuário já esteja conectado e deseje se conectar novamente.

Além das exceções de validação, existe uma exceção que deve ser tratada pelo desenvolvedor caso não seja possível se conectar ao servidor. Essa exceção é chamada de

NaoPossivelConectarException. Quando essa exceção é lançada, o problema não se encontra exatamente no cliente. Essa exceção só é chamada quando, por algum motivo, o framework não está conseguindo se conectar no servidor - normalmente, porque existe um firewall configurado ou porque o jogador passou um ip inválido.

Além das exceções tratadas pelo desenvolvedor, existe uma exceção especial tratada pelo framework. Seu nome é CallException e se encontra no pacote n". Essa exceção pode ser lançada por qualquer um dos métodos de chamada de procedimento remoto tanto no cliente quanto no servidor.

Se essa exceção for lançada, indica que por algum motivo o método não está sendo executado no servidor. Dentre as possíveis razões, o método pode não existir ou seu acesso pode ter sido bloqueado. Se isso ocorrer, o framework automaticamente desconecta o cliente do servidor. Foi decidido proceder dessa forma, pois, não seria interessante continuar com o cliente conectado se um dos métodos não estivesse operando.

Como pode ser observado no diagrama, a classe Proxy não possui o objeto da classe DualRpcClient. Essa classe é a classe que liga o framework ao servidor. Para que a conexão ocorra, uma série de detalhes de configuração precisam ser feitos na instancia de DualRpcClient. Como a função principal do proxy é encaminhar as mensagens do servidor, decidiu-se por utilizar uma outra classe, chamada de Cliente, que acumula a função de configurar e utilizar a instancia de DualRpcClient.

6.2 Servidor invocando métodos no cliente

Até agora foram analisadas as formas que permitem ao proxy se conectar e chamar métodos no servidor. A partir deste ponto é analisado o processo contrário, o servidor chamando um método no cliente.

Assim como o servidor possui uma classe tratadora que corresponde aos métodos que podem ser acessados pelo cliente, o framework também possui uma classe que possui os métodos que serão chamados pelo servidor. Essa classe é chamada de ClienteTratador. Quando o cliente se conecta ao servidor através do framework, uma instância dessa classe é criada e associada à instância de DualRpcClient.

Essa instância pode ser chamada, então, a qualquer momento da execução pelo servidor. Ela possui os métodos de resposta do servidor a eventos do cliente. A maioria dos métodos utilizados pelo framework são assíncronos, o que possibilita que o cliente não fique esperando uma resposta do servidor. Assim, caso o servidor esteja congestionado ou por algum motivo demorando para responder, o cliente não correrá o risco de ficar com uma thread do seu programa parada esperando pela resposta do servidor.

A única forma de a mensagem assíncrona ser perdida é, portanto, se a chamada não for efetuada, o que irá lançar uma CallException, ou se o servidor não puder ser acessado, o que irá automaticamente desconectar o cliente.

Garantindo que a mensagem foi entregue ao servidor, basta que ele a processe e dependendo da mensagem, chame métodos no cliente através da instância de ClienteTratador.

O cliente pode desejar tratar em mais de um lugar essa mensagem. Ele pode querer tratá-la no modelo e na interface ao mesmo tempo, por esse motivo foi usado o padrão *Observer*. [GAM 1994]

Se o cliente quiser receber essa mensagem do servidor ele precisa implementar a interface OuvidorProxy. As instâncias dessa classe devem se cadastrar no Proxy através do método Proxy.addOuvinte(OuvidorProxy ouvidor).

Como o proxy que possui ouvintes e não a classe ClienteTratador, a classe ClienteTratador passa ao Proxy as mensagens recebidas do servidor e o Proxy as encaminha para os clientes.

6.3 Adaptando Jogos ao framework

Foram adaptados inicialmente 3 jogos: um jogo da velha, um jogo de xadrês e um jogo chamado BagChall.

A principal dificuldade encontrada foi entender a lógica dos jogos tratados, após vencido esse obstáculo, bastou definir como eles deveriam se comunicar com o framework. Devido ao prévio conhecimento do funcionamento do framework isso foi realizado de forma rápida. E, a partir do primeiro jogo adaptado, o desenvolvimento das posteriores se mostrou ainda mais simples.

A análise completa da adaptação de um jogo segue no próximo capítulo.

7. Adaptando um jogo stand-alone

A seguir se encontram os procedimentos para adaptar qualquer jogo de não tempo real ao servidor. Como exemplo foi adaptado um jogo da velha construído pelo professor Ricardo Pereira e Silva para as aulas de Análise e Projeto Orientados a Objetos.

O jogo se encontra para download na página: http://www.inf.ufsc.br/~ricardo/multiJogos/jogos/jogoVelha/standAlone. Junto com o jogo existe a documentação final onde se encontram diversos diagramas.

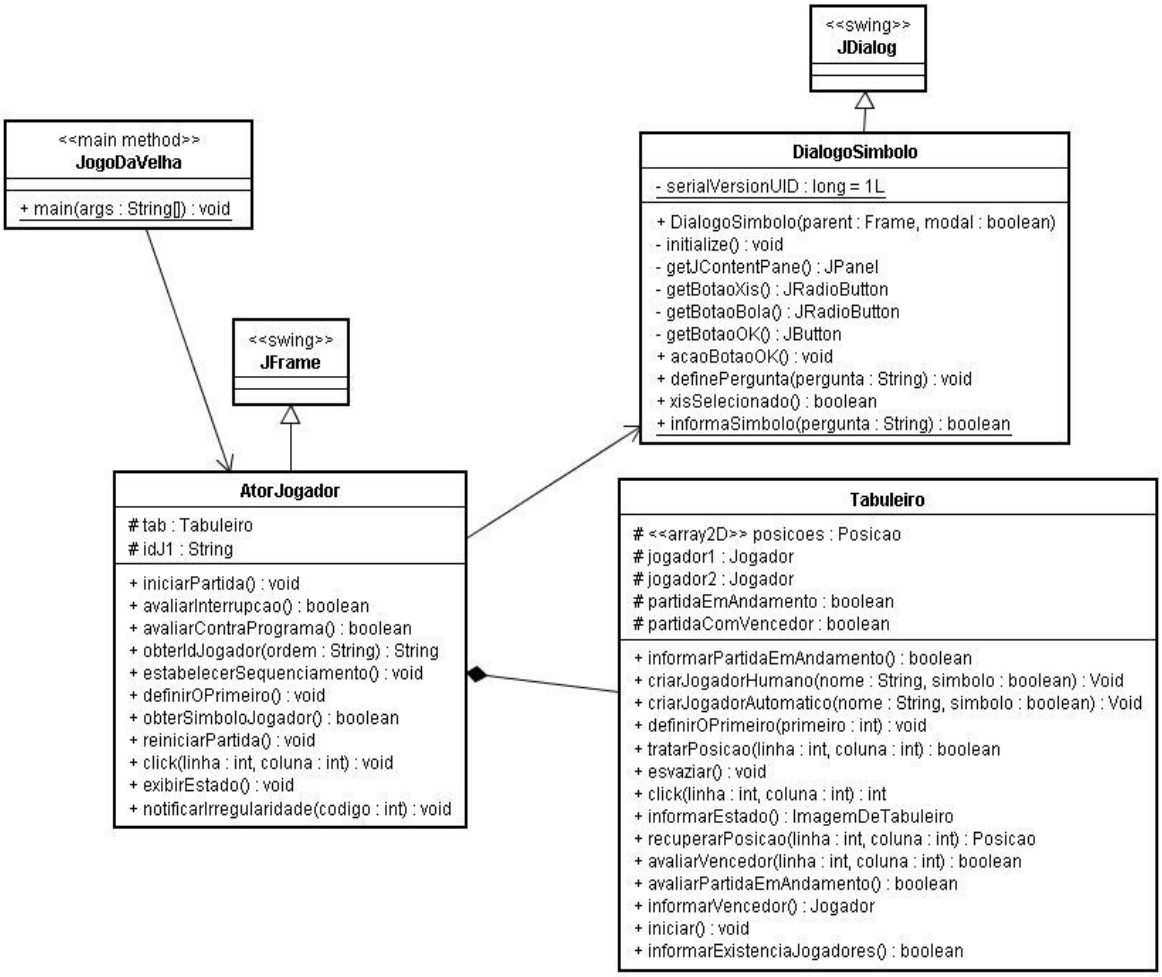

Figura 7.1 Diagrama de classes(reduzido) do Jogo da Velha

A figura 7.1 apresenta parte do diagrama de classes do jogo da velha, essa parte do diagrama corresponde ao modulo de interface gráfica e parte do modelo do jogo.

O *AtorJogador* é a principal classe, ele controla os itens da interface gráfica[14] e possui os tratadores para os eventos vindos da interface e que se refletem no modelo. Como pode ser visto ele estende um JFrame, uma classe do pacote swing responsável por encapsular todos os elementos de uma GUI.

[14] Ele estende *JFrame* e possui menu, label's, etc... Além disso também implementa a interface *ActionListener* e é ouvidor dos elementos da interface gráfica

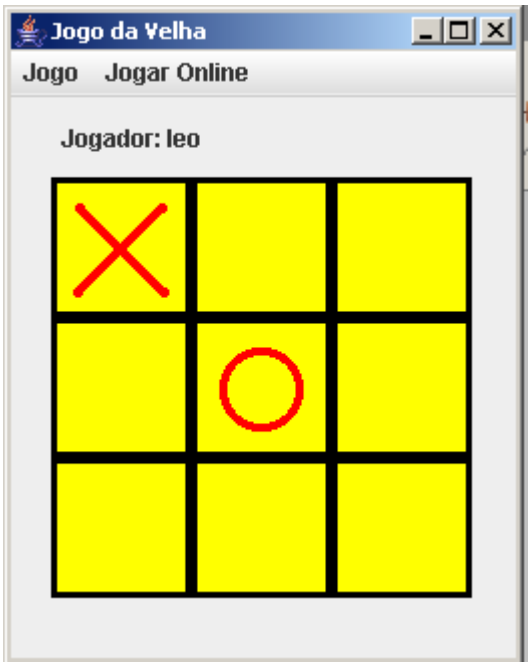

Figura 7.2 Interface Gráfica do jogo da velha

A figura 7.2 mostra a interface gráfica do jogo da velha, basicamente ela apresenta o nome do jogador da vez e o estado do jogo até o momento. O estado é representado em uma matriz 3x3 marcada com o símbolo do jogador, por padrão xis ou bola.

O funcionamento do jogo da velha é bastante simples, cada jogador possui um símbolo único e tem uma vez de jogar. Na sua vez ele deve marcar uma das casas com o seu símbolo, se algum dos dois conseguir formar uma cadeia de três casas consecutivas com o seu símbolo ele é o vencedor[15], caso nenhum dos dois realize o feito, até que o tabuleiro esteja totalmente completado, então o jogo é finalizado com empate, popularmente chamado de "Velha".

Qualquer jogo que desejar futuramente utilizar o ambiente[16] deve implementar basicamente: como chamar os métodos no servidor e como ser chamado pelo servidor.

Utilizando o framework fornecido esses pontos de implementação já estão definidos. Qualquer método chamado no servidor deve utilizar uma instância da classe Proxy, e qualquer método chamado pelo servidor no jogo é repassado aos objetos que implementarem a interface OuvidorProxy[17] e se registrarem no Proxy.

[15] Essa cadeia pode ser na horizontal, vertical ou em uma das diagonais principais.

[16] Entenda-se por ambiente o conjunto servidor e framework.

[17] O parte do nome "Ouvidor" se refere ao padrão Observer [GAM 1994]

Os métodos invocados geralmente correspondem aos eventos gerados por um jogador e que deve ser transmitido aos outro jogadores para que possam atualizar o estado dos seus jogos.

O ponto de entrada e saída no jogo da velha foi definido em um único objeto que é conhecido e conhece o AtorJogador. Esse objeto foi chamado de *AtorRede*.

AtorRede chama métodos do proxy e por implementar a interface OuvidorProxy também recebe chamadas dele.

Apesar dessa divisão, o que ocorre no processo de adaptar o framework ao projeto é uma junção de informações. Para cada método chamado no servidor, corresponderá a um método chamado no cliente pelo servidor. Esses métodos podem ser agrupados em três "subprocessos": conectando, jogando e desconectando.

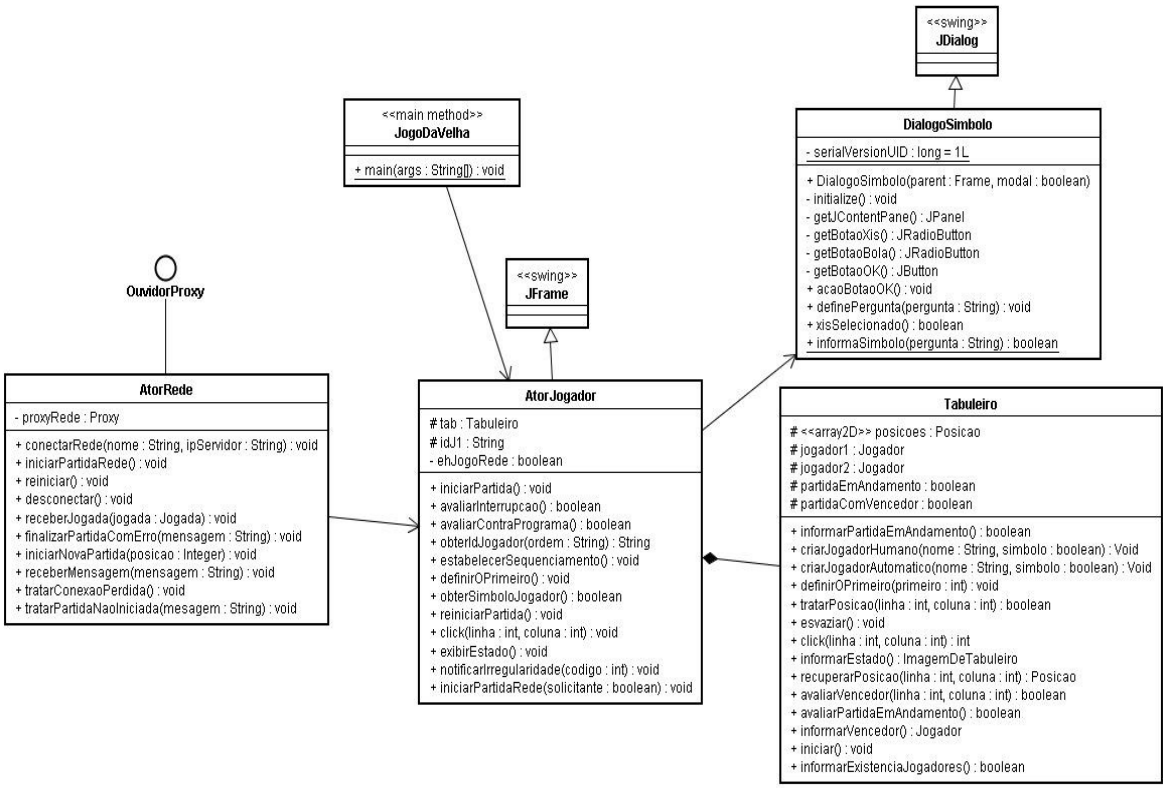

Figura 7.3 Diagrama de classes do Jogo da Velha Multi-jogador

O jogo foi, portanto, modelado como na figura 7.3, esse modelo possui além do diagrama da figura 7.1 a presença do *AtorRede*, esse é o ator que foi designado por receber as informações do servidor e passar para o jogo e também de passar as informações do jogo para o servidor através do framework.

São necessários incluir dois arquivos no jogo a ser adaptado. O primeiro é o .jar com o framework. O download do framework é feito através do site: http://java.inf.ufsc.br/games.

O segundo é um arquivo de configuração do jogo e é disponibilizado ao cadastrar o jogo no site: http://java.inf.ufsc.br/games. Esse arquivo se chama "jogoMultiPlayer.properties" e deve ser colocado dentro do classpath da aplicação

7.1 Conectando

Para se conectar ao servidor o cliente/aplicativo precisa fazer uma chamada ao servidor. Todas as chamadas ao servidor são realizadas pelo framework. Para realizar qualquer comunicação o cliente deve utiliza um Proxy. Esse Proxy implementa o padrão Singleton[GAM 1994].

Para conectar-se ao servidor o cliente/aplicativo realiza a chamada ao método conectar, como segue na figura 7.4:

```
Proxy.getInstance().conectar(ipServidor, nome);
```

Figura 7.4 Código para se conecta ao servidor

Nessa chamada o jogo precisa passar dois parâmetros do tipo String, o primeiro, ipServidor, corresponde ao ip do servidor de jogos. No nosso caso o servidor de jogos está rodando hoje em na máquina saturno.inf.ufsc.br, um dos servidores da UFSC e o usuário pode entrar com o nome de host (saturno.inf.ufsc.br) ou com o ip da máquina na rede (150.162.60.33).

Caso a entrada nesse caso seja nula ou vazia o framework está configurado para se conectar no servidor anteriormente citado automaticamente.

O segundo parâmetro, nome, serve apenas como identificador do jogador para os outros jogadores adversários.

Há, porém alguns erros que podem acontecer nessa chamada de método. Dentre eles se encontram: o servidor pode estar fora do ar, a rede em que o cliente se encontra pode bloquear acesso a todas as portas, o cliente/aplicativo já está conectado no servidor. Esses erros foram tratados através de exceções que são lançados pelo framework caso eles ocorram. Por parte do cliente basta definir quando e de que forma essas exceções serão tratadas.

São elas: JahConectadoException e NaoPossivelConectarException.

```
try {
      Proxy.getInstance().conectar(ipServidor, nome); //1
} catch (ArquivoNaoExisteException e) { //5
      e.printStackTrace();
      JOptionPane.showMessageDialog(atorJogador,
      e.getMessage()); //4
} catch (JahConectadoException e) { //2
      e.printStackTrace();
      JOptionPane.showMessageDialog(atorJogador,
      e.getMessage()); //4
} catch (NaoPossivelConectarException e) { //3
      e.printStackTrace();
      JOptionPane.showMessageDialog(atorJogador, "Erro: " +
      e.getMessage());
}
```

Figura 7.5 Código de tratamento de exceções para o método conectar

Pode ser observado analisando a figura 7.5:

1. Método que tenta conectar o cliente no servidor, já discutido anteriormente.
2. Captura a exceção JahConectadoException.
3. Captura a exceção NaoPossivelConectarException.
4. Exibe uma mensagem ao usuário, nesse caso exibe no gui a mensagem de exceção
5. Captura a exceção ArquivoNaoExiteException, caso o arquivo de configuração não esteja presente no classpath.

Caso nenhuma dessas exceções seja lançada o cliente pode considerar-se cadastrado no servidor.

7.2 Jogando

Para poder jogar é preciso que o jogador já esteja conectado e já tenha passado pela fase anterior. Nesse momento o servidor já conhece o cliente e já existe uma "ponte" de comunicação entre ambos. Para se comunicar com o servidor o cliente se utiliza do Proxy visto anteriormente, a partir de então o proxy processa informações e dependendo do caso envia informações para os clientes e pode chamar mensagens do framework. As mensagens chamadas pelo servidor são encaminhadas ao proxy que as encaminhas a seus ouvidores, completando assim o ciclo do padrão Observer[GAM 1994]
. Um ouvinte do framework pode ser qualquer objeto Java que implemente a interface OuvidorProxy disponibilizada pelo framework, para implementar uma interface Java duas coisas são necessárias. A primeira é adicionar a declaração "*implements OuvidorProxy"* a declaração da classe.

A partir daí para que o código seja comilado é necessário que os métodos da interface sejam implementadas pela classe AtorRede. Esses métodos serão discutidos a seguir.

Para que o proxy conheça a classe AtorRede é necessário que ela se cadastre no proxy, o código está presente figura 7.6.

```
Proxy.getInstance().addOuvinte(this);
```

Figura 7.6 Código para adicionar um ouvinte no proxy

Agora que já temos a comunicação entre servidor e cliente estabelecida, passemos para a próxima fase, a de como enviar e receber informações e realizar os jogos.

7.3 Iniciando uma Partida

Sempre que um ou mais jogadores desejarem trocar informações uma partida é iniciada no servidor. Uma partida mantêm uma sessão aberta onde diversos usuários podem enviar e receber requisições para/de outros usuários.

A partir do momento que está conectado o cliente iniciar uma partida de duas formas, de maneira passiva ou ativa. Na forma ativa o cliente que requisita o início de uma

partida e outros usuários são escolhidos aleatoriamente para iniciarem a mesma partida, esses usuários escolhidos iniciam a partida de maneira passiva.

Para iniciar a partida de maneira ativa o cliente chama o seguinte método da instancia de Proxy:

```
try {
      Proxy.getInstance().iniciarPartida(2);
} catch (NaoConectadoException e) {
      e.printStackTrace();
      JOptionPane.showMessageDialog(atorJogador,
      e.getMessage());
}
```

Figura 7.7 Código para iniciar uma partida de 2 jogadores

Pode ser observado analisando a figura:

1. O método inciarPartida inicia uma nova partida no servidor, o servidor sorteia participantes que serão os adversários do jogador. A quantidade de jogadores de uma partida é passada como parâmetro. Nesse caso (Jogo da Velha) serão dois jogadores por partida, o jogador que está iniciando a partida mais o jogador adversário.
2. Captura a exceção NaoConectadoException, essa exceção indica que o cliente não pode iniciar uma partida nesse momento pois ainda não se encontra conectado no servidor.
3. Uma das formas de tratar a exceção exibindo uma mensagem para o usuário.

Os jogadores selecionados pelo servidor para participar da mesma partida recebem uma chamada no método *iniciarNovaPartida(Integer posicao)*. Esse é um dos métodos que deve ser implementado por um OuvidorProxy. O parâmetro posição indica qual a ordem dos jogadores, inicia em 1 (primeiro jogador que por padrão é quem requisitou a nova partida) até o último jogador da partida. Caso não existam jogadores suficientes ou por algum outro motivo a partida não possa ser iniciada método tratarPartidaNaoIniciada(String message) é chamado nos ouvintes.

7.4 Enviando e Recebendo Jogadas

Nesse momento já existe uma partida iniciada no servidor entre dois ou mais jogadores. Para enviar qualquer tipo de mensagem para o outro lado da comunicação foi estabelecido apenas um par de métodos. Um que envia e outro que recebe.

A principal característica, que os torna genéricos, é o fato de qualquer tipo de objeto ser passado de um lado para o outro, desde de que esse objeto estenda a interface *Jogada.*

Essa interface não possui métodos, é apenas uma interface de marcação que estende outra interface de marcação chamada *Serializable*. Para que um objeto seja enviado pela rede ele deve ser antes serializado. Cabe ao desenvolvedor apenas garantir que a classe que ele deseja mandar implemente a interface Jogada, se essa classe possuir alguma associação, essas associação deve também implementar a interface Jogada[18], e assim por diante.

Ao analisar o jogo da velha observou-se que a única informação passada para o outro lado era referente ao símbolo colocado em uma determinada casa. Esse símbolo é

[18] Para as classes de associação também pode ser utilizada a interface Serializable ao invés de Jogada.

constituído de dois campos inteiros, linha e coluna. O campo linha representando as 3 linhas horizontais e o campo coluna representando as 3 verticais.

Foi criado uma classe chamada JogadaVelha que era constituída apenas de dois campos int, um para linha e outro para coluna, e a classe implementa a interface Jogada provinda do framework.

Quando um jogador, sabendo que é sua vez de jogar, aperta um dos nove botões, ele passa a informação para o jogo da casa onde deve ser colocado o seu símbolo. O jogo captura essa informação através do método click() do AtorJogador.

Existem diversas formas de modificar esse método para enviar uma mensagem para a rede. O desenvolvedor pode encaminhar a chamada para o AtorRede, pode sobrescrever o método em uma subclasse, pode simplesmente modificar o método, etc...

Apenas por facilidade de desenvolvimento o método foi apenas modificado através da inserção de código. Esse código (apresentado na figura 7.8) apenas confere que o jogo é um jogo multi-jogador[19] e se é a vez do jogador local, o jogador que está rodando o aplicativo e portanto o método.

Caso tudo isso esteja dentro dos conformes uma JogadaVelha é instanciada e enviada para o outro lado através do método enviaJogada(Jogada jogada).

Esse método é assíncrono e existe a garantia de que a mensagem foi recebida pelos outros jogadores. Essa garantia é dada pelo framework.

```
if (ehJogoRede && (tab.getJogador2().informarDaVez())) {
      Jogada jg = new JogadaVelha(linha, coluna);
      try {
            Proxy.getInstance().enviaJogada(jg);
      } catch (NaoJogandoException e) {
            e.printStackTrace();
            JOptionPane.showMessageDialog(this, e.getMessage());
      }
}
```

Figura 7.8 Enviando uma Jogada

Quando um jogador enviar uma mensagem, ela é encaminhada através do servidor para todos os outros jogadores que estejam na mesma partida. Para receber uma jogada de outro jogador basta implementar o método receberJogada(Jogada jogada) do OuvidorProxy.

Sempre que o servidor encaminha uma mensagem, esse método é ativado em todos os ouvidores devidamente registrados no Proxy. Se em um dos ouvidores esse método não fazer nada, basta deixar o método em branco.

No jogo da velha é realizado o processo inverso ao envio de uma jogada. Como apenas um tipo de Jogada é enviado pela rede, é sempre do tipo JogadaVelha, então é feito um cast. O método da figura 7.9 foi implementado no AtorRede, o único OuvidorProxy.

Caso mais de uma classe implementasse a interface Jogada, seria necessário verificar qual a instancia através de um *instanceof* (figura 7.10).

```
public void receberJogada(Jogada jogada) {
      JogadaVelha jg = (JogadaVelha) jogada;
      atorJogador.click(jg.getLinha(), jg.getColuna());
}
```

[19] A variável ehJogoRede é um boolean. Quando seu valor é true o jogo é multi-jogador

Figura 7.9 Recebendo uma Jogada

```
public void receberJogada(Jogada jogada) {
      if (jogada instanceof JogoDaVelha){
      JogadaVelha jg = (JogadaVelha) jogada;
      atorJogador.click(jg.getLinha(), jg.getColuna());
}
```

Figura 7.10 Recebendo uma Jogada e verificando o tipo

7.5 Desconectando

Essa etapa é bastante simples. Caso o jogador escolha por fechar o jogo, o servidor reconhecerá que falta um cliente no jogo e automaticamente desconectará aquele cliente. Caso o jogador deseje se desconectar e voltar a jogar localmente existem um método chamado desconectar, esse método assim como os outros anteriormente explicados se encontra na classe Proxy e pode ser chamado como na figura a seguir.

```
try {
      Proxy.getInstance().desconectar(); // 1
} catch (NaoConectadoException e) { // 2
      e.printStackTrace();
      JOptionPane.showMessageDialog(atorJogador, "Erro:
      "+e.getMessage());
)
```

Figura 7.11 Desconectando um cliente

Pode ser analisado observando a figura 7.11:

1. Procedimento chamado para desconectar um cliente do servidor
2. Tratando a exceção NaoConectadoException

O jogo adaptado, junto com documentação se encontra no site: www.inf.ufsc.br/~ricardo/multiJogos/jogos/jogoVelha/standAlone.

8. Conclusão

O servidor multi-jogos produzido permite conectar diversos jogadores por meio de uma rede, para realização de partidas. O framework produzido disponibiliza o acesso ordenado ao servidor e controla o envio e recepção de mensagens no cliente, tornando aspectos de comunicação transparentes ao usuário.

A validação dos artefatos de software produzidos foi realizada conectando-se diferentes jogos desenvolvidos ao servidor, e os jogadores realizando diferentes partidas simultâneas.

8.1 Realizações

O trabalho desenvolvido teve como objetivo principal a construção de um servidor multi-jogos para jogos não tempo real para uso dos alunos da disciplina de Análise e Projeto Orientados a Objetos, na implementação dos seus trabalhos finais.. Ele compreendeu também a construção do framework de acesso ao servidor.

Para tanto, foi necessário:

- Projetar um protocolo de alto nível para comunicação entre os jogadores de um jogo.
- Entender arquitetura cliente/servidor.
- Estudar tecnologias de computação distribuída em geral.
- Estudar tecnologias Java para sistemas distribuídos.
- Construir um framework orientado a objetos.
- Construir um servidor de pequeno porte.
- Construir artefatos de software reutilizáveis.
- Construir um ambiente de desenvolvimento genérico.
- Adaptar três jogos NRT ao sistema desenvolvido. Um jogo da velha, um jogo de Xadrez e um jogo chamado Bagchall.

8.2 Limitações

A principal limitação do servidor é não conseguir evitar trapaças. Para que o servidor pudesse evitar trapaças era necessário que ele conhece a lógica dos jogos tratados.

Outra limitação, já citada, é o fato de o servidor aceitar teoricamente um máximo de 1500 jogadores ao mesmo tempo, estando eles em qualquer um dos jogos conectados. Essa limitação é uma característica do framework DualRPC, porém não foram feitos ensaio que garantam ser possível chegar a esse limite, considerando outras limitações inerentes ao ambiente em que o servidor está em execução.

8.3 Trabalhos Futuros

Diversos aspectos podem ser adicionados aos artefatos de software produzidos, de forma a ampliar e melhorar oque já está construído. Entre eles estão:

- Adaptar a arquitetura do servidor, de forma a permitir a existência de vários servidores que se comuniquem em *peer to peer*, como acontece nos jogos desenvolvidos atualmente, para melhorar a escalabilidade;
- Tornar o servidor utilizável para jogos escritos em outras linguagens de programação;
- Construir um portal para o desenvolvedor de forma a poder controlar o jogo em tempo real;
- Adaptar o framework DualRPC para funcionar sob tecnologia J2ME, possibilitando o uso em jogos em dispositivos móveis.
- Estender as funcionalidades para outros tipos de jogos.

8.4 Considerações Finais.

Na concepção e desenvolvimento desse trabalho, conseguiu-se alcançar os objetivos propostos inicialmente. Existe uma versão em funcionamento do servidor de jogos e do framework apta a ser utilizada pelos alunos da disciplina de Análise e Projeto Orientados a Objetos na concepção de seus jogos.

Esses artefatos produzidos garantem a conexão de diferentes jogos sem exigência de tempo real, todos operando ao mesmo tempo.

9. Referências Bibliográficas

[AVO 2007] AVONDOLIO, Donald et al. **Professional Java JDK, 6ª ed.** Wiley Publishing, Inc, Indianapolis, Indiana , 2007.

[CEC 2004] CECION, F. R. **Suporte a jogos distribuídos e de tempo real**. Semana acadêmica, UCS. Disponível em: <http://ggpd.inf.ufrgs.br/wili/upoads/FreeMMG.GreeMMG/Palestra_UCS_7.pdf>. Acesso em 2 out. 2007.

[DEI 2004] DEITEL, H. M; DEITEL, P. J. **Java How To Programm, 6ª ed.** Prentice Hall PTR, agosto de 2004.

[FEI 2003] FEIJÓ, Bruno; KOZOVITS, Luaro Eduardo. **Arquitetura para Jogos Massive Multi-Player.** Artigo. PUC. Rio de Janeiro, 2003.

[GAM 1995] GAMMA Erich, et al. **Design Patterns Elements of Reusable Object-Oriented Software**. Addison-Wesley Pub, Jan. 1995.

[HOR 2004a] HORSTMANN, CAY; CORNELL, GARY. **Core Java Volume I – Fundamentals, 7ª ed.** Prentice Hall PTR, novembro 2004.

[HOR 2004b] HORSTMANN, CAY; CORNELL, GARY. **Core Java Volume II – Advanced Features, 7ª ed.** Prentice Hall PTR, novembro 2004.

[JOR 2006] JORGE A. N. S. Anibolete, Tulio **Algorítimos Distribuídos em jogos Multi-usuáiros**. Monografia. PUC. Departamento de Informática e Estatística. Rio de Janeiro, 2006.

[LAR 2000] LARMAN, Craig. **Utilizando UML e Padrões: Uma introdução á análise e ao projeto orientados a objetos.** Bookman 2000.

[MEL 2007] MELO, Paulo Henrique Borges. **Invocação Remota de Métodos (RMI) no TIGER**. Disponível em: http://www.javafree.org/content/view.jf?idContent=72. Acesso em 22 fev 2007.

[NUN 2003] NUNES, Leonardo. Sokects em Java. **Mundo Java**, Rio de Janeiro, n. 2, p. 29-32, 2003.

[ORF 1996] ORFALI, Robert; Et al. **Cliente/Servidor – Guia Essencial de Sobrevivência.** INFOBOOK S. A. 1996.

[PED 2006] PEDROSA, D. V. **Jogos Multiplataforma Mutliusuário.** Recife, out 2006.

[SIE 2005] SIERRA, KATE; SIERRA, BERT. **Head First Java**, 2ª ed. Alta Books Ltda, 2005.

[SUN 2007] **Remote Method Invocation (RMI).** Disponível em: <http://Java.SUN.com/javase/6/docs/technotes/guides/rmi/index.html>. Acesso em 19 fev. 2007.

[SIL 2000] SILVA, R. P. **Suporte ao desenvolvimento e uso de frameworks e componentes**. Porto Alegre: UFRGS/II/PPGC, mar. 2000. Tese de Doutorado.

[WIR 1990] WIRFS-BROCK, R.; JOHNSON, R. E. **Surveying current research in object-oriented design**. Communications of the ACM, New York, v.33, n.9, Sept. 1990.

Anexo I: Glossário

Jogador ou Cliente - Aplicação desenvolvida que usa o framework.

Servidor - Programa que recebe e trata as requisições dos clientes.

Jogo - Sessão do servidor compartilhada entre todos os jogadores/cliente de um mesmo jogo/cliente.

Jogada - Qualquer mensagem trocada entre os jogadores na realização de um jogo e que de alguma forma mude o estado do jogo ou tenha um significado para os outros jogadores.

Partida - Sessão compartilhada entre usuários um ou mais usuários de um mesmo jogo que estão trocando mensagens diretamente. As informações de um jogador são repassadas para todos os outros. Um jogador só pode participar de uma partida por vez.

Adversário(s) - Jogadores que estão jogando o mesmo jogo, participando da mesma partida e que estão jogando contra o jogador/cliente.

Anexo II: Casos de Uso

Casos de Uso do servidor de jogos

Atores: Ator Cliente, Ator Servidor.

Caso de Uso 001 - Conectar Cliente

Ator principal: Cliente.
Ator secundário: Servidor.
Pré-requisito:

Fluxo Principal:
1. O ator envia uma mensagem de conexão ao servidor, informando o nick, o id do jogo no qual deseja se conectar.
2. O servidor cria um novo jogo baseado no id.
3. O ator é inserido nesse jogo.
4. O servidor envia uma mensagem ao cliente informando que a conexão foi estabelecida

Fluxo Alternativo e Exceção:
1) O jogador por algum motivo de rede não consegue se conectar.
    1. É exibida uma mensagem para o usuário ("Por algum motivo não é possível se conectar. Por favor tente novamente mais tarde)

2) Jogo já existe.
    1. Ir para 3.

Caso de Uso 002 - Iniciar Nova Partida/Reiniciar Partida

Ator principal: Cliente
Ator secundário: Servidor
Pré-requisito: Ator já deve estar conectado (caso de uso 001)

Fluxo Principal:
1. O ator requisita uma nova partida passando o número de jogadores na partida.
2. O servidor envia a requisição para o jogo no qual o jogador está inserido.
3. O jogo seleciona clientes para participar do jogo.
4. O jogo cria uma nova partida com esses jogadores.
5. O jogo manda a partida iniciar.
6. A partida envia uma mensagem de partida iniciada a todos os jogadores.

Fluxo Alternativo e Exceção:
3) Não existem adversários suficientes
    1. O jogo envia uma mensagem aos clientes ("Não existem jogadores suficientes para criar uma nova partida").

Caso de Uso 003 - Aguardar Partida

Ator principal: Servidor
Ator secundário: Cliente,
Pré-requisito: Ator já deve estar conectado(caso de uso 001) e um outro jogador solicitou o inicio de uma partida (caso de uso 002) e o cliente (ator secundário) foi selecionado para a partida.

Fluxo Principal:
1.O serivdor envia uma requisição informando que o cliente foi alocado em uma nova partida.
2.O cliente cria um novo jogo.

Caso de Uso 004 - Efetuar Jogada

Ator principal: Cliente
Ator secundário: Servidor.
Pré-requisito: Jogado já deve estar em uma partida (caso de uso 002 ou caso de uso 003)

Fluxo Principal:
1. O cliente envia uma mensagem ao servidor.
2. O servidor trata essa mensagem enviando a todos os outros clientes que estão participando de uma mesma partida.

Caso de Uso 005 - Receber Jogada

Ator principal: Cliente
Ator secundário: Servidor.
Pré-requisito: Jogado já deve estar em uma partida (caso de uso 002 ou caso de uso 003) e um outro jogador deve ter efetuado uma jogada (caso)

Fluxo Principal:
1. O cliente recebe a jogada.
2. O cliente atualiza a jogada no jogo.
3. O cliente envia uma mensagem de jogada recebida ao servidor (?).

Caso de Uso 006 - Finalizar Partida (quando alguém ganha)
Ator principal: Cliente
Ator secundário: Servidor.
Pré-requisito: Uma partida deve estar em andamento no servidor.

Fluxo Principal:
1. O cliente requisita a finalização da partida na qual se encontra ao servidor.
2. O servidor repassa a requisição para a partida.
3. A partida envia uma mensagem a todos os jogadores que a partida foi finalizada.
4. A partida recadastra os jogadores no jogo.

Fluxo Alternativo e Exceção:

Caso de Uso 007 - Desconectar
Ator principal: Cliente
Ator secundário: Servidor.
Pré-requisito: O jogador deve estar conectado a um jogo e pode estar em uma partida.

Fluxo Principal:
1. O cliente requisita sua desconexão de um jogo.
2. O servidor repassa a informação para o jogo.
3. O jogo verifica que o cliente está em alguma partida e envia uma mensagem de finalização de partida.
4. A partida é finalizada.

Fluxo Alternativo e Exceção:

Caso de Uso 08 - Tratar jogador que caiu

Ator principal: Servidor
Ator secundário: Servidor.
Pré-requisito: Um dos jogadores se finalizou sua sessão com o servidor.

Fluxo Principal:
1. O servidor envia uma mensagem para o jogo de que o jogador não está mais conectado.
2. O jogo verifica que o cliente está em alguma partida e envia uma mensagem de finalização de partida.
3. A partida é finalizada.

Fluxo Alternativo e Exceção:
- 2) O jogador não estava em nenhuma partida.
- Fim do caso de uso.