UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

PROTÓTIPO DE MOTOR DE SERVIDOR DE JOGOS *ONLINE* EM MASSA

DANIEL PRESSER

BLUMENAU
2006

2006/2-06

DANIEL PRESSER

PROTÓTIPO DE MOTOR DE SERVIDOR DE JOGOS *ONLINE* EM MASSA

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

Prof. Francisco Adell Péricas, Ms - Orientador

BLUMENAU
2006

2006/2-06

PROTÓTIPO DE MOTOR DE SERVIDOR DE JOGOS *ONLINE* EM MASSA

Por

DANIEL PRESSER

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

Presidente:	Prof. Francisco Adell Péricas, Ms – Orientador, FURB
Membro:	Prof. Maurício Capobianco Lopes, Ms – FURB
Membro:	Prof. Sérgio Stringari, Ms – FURB

Blumenau, 1 de dezembro de 2006

Dedico este trabalho aos meus familiares, amigos, professores e colegas que me apoiaram e contribuíram, direta ou indiretamente, para a conclusão deste trabalho.

AGRADECIMENTOS

À Alexandra, por todo o apoio e por estar ao meu lado até a conclusão deste trabalho.

À minha família, que me possibilitou iniciar o curso, além de despertar em mim a vontade da busca pelo conhecimento.

Aos meus colegas, tanto pela ajuda na implementação deste trabalho, quanto pelo convívio, que tornou este período mais agradável.

Ao Odair e à Simple Tecnologia, pelo apoio e folgas providenciais.

Ao meu orientador, Francisco Adell Péricas, pelo apoio e por ter acreditado na conclusão deste trabalho.

Algo é só impossível até que alguém duvide e acabe provando o contrário.

Albert Einstein

RESUMO

O presente trabalho descreve o desenvolvimento de um protótipo de software genérico para utilização na parte servidora de jogos *online* em massa no estilo RPG, popularmente conhecidos por MMPGs, através da utilização de métodos e tecnologias visando otimizar o grande volume de processamento e conexões que este tipo de aplicação requer, bem como garantir sua extensibilidade através da utilização de uma linguagem de *scripts*. Apresenta-se a especificação e implementação do protótipo desenvolvido, que utiliza o modelo de camadas proposto por Lyra Studios em conjunto com o modelo publicador-assinante para implementação do servidor, utilização de IOCP para gerenciar as conexões e a linguagem Lua para *scripts* de extensão.

Palavras-chave: Jogos *online*. IOCP. Redes. Linguagem de *scripts*.

ABSTRACT

This paper describes the development of a generic software archetype to be used in RPG style massive mutiplayer online games (also known as MMPGs) server side, through the use of methods and tecnologies aiming to optimize the large amount of processing and the connection number that this application type requires, as well as to guarantee its extensibility through the use of a scripting language. It's presented the project's specification and implementation, wich uses the layers model proposed by Lyra Studios, with the publisher-subscriber model to the server's implementation, IOCP for connection management, and Lua scripting language for extensions.

Key-words: Online games. Computer networks. Scripting languages.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

BNF – *Backus Naur Form*

CP – *Completion Port*

IGDA – *International Game Developers Association*

IOCP – *Input/Output Completion Port*

IP – *Internet Protocol*

MMORPG – *Massive Multiplayer Online Role Playing Game*

MMP – *Massive MultiPlayer*

MMPG – *Massive MultiPlayer Game*

RPG – *Role Playing Game*

SO – Sistema Operacional

TCP – *Transmission Control Protocol*

UDP – *User Datagram Protocol*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Os jogos *online* em massa, ou *Massive MultiPlayer Games* (MMPGs), representam um dos mais inovadores ramos na indústria do entretenimento eletrônico. Trata-se de jogos construídos para serem utilizados exclusivamente *online*. Em geral, são baseados num universo virtual que simula o mundo real, porém, seguindo um enredo e história pré-definidos. Além disso, MMPGs costumam seguir o estilo de jogos de interpretação de personagens, ou *Role Playing Game* (RPG). Neste formato, cada jogador interpreta um personagem baseado numa gama de tipos, ou classes, pré-configuradas. O objetivo do jogador é desenvolver as habilidades específicas de seu personagem, interagindo com o universo virtual do jogo e com os outros jogadores.

Um exemplo de enredo de RPG são os baseados em histórias medievais. Nestes casos, os jogadores devem desenvolver personagens como espadachins, arqueiros, mercadores, etc. Cada uma destas classes de personagens tem algumas habilidades mais importantes que outras, como um espadachim que precisa mais de força do que inteligência, ou um arqueiro que necessita de maior destreza do que vitalidade. O jogo consiste, então, em desenvolver essas habilidades através de lutas e treinamentos oferecidos pelo enredo.

Diz-se que MMPGs são em massa pelo número de jogadores que podem interagir simultaneamente num mesmo mundo. Segundo a *Internation Grame Developers Association* (IGDA) (2003, p. 10, tradução nossa), “MMPGs caracterizam-se por milhares de jogadores (entre 2000 e 4000 nos servidores mais atuais, com alguns jogos necessitando de vários servidores) jogando simultaneamente num grande mundo persistente”. Com um volume tão grande de jogadores, cria-se uma espécie de sociedade, em que os jogadores podem se reunir em grupos onde cada personagem executa seu papel específico e acaba por beneficiar a si próprio e ao grupo inteiro.

Este novo formato, onde milhares de pessoas compartilham um mesmo servidor, leva ao limite o conhecimento que se tinha de ambientes *online* para jogos, pois se trata de um segmento que conta com pouco tempo de pesquisa, se comparado a outros setores, como a computação gráfica e o áudio. Os primeiros verdadeiros MMPGs começaram a aparecer em meados da década de 1990, porém sua popularização realmente aconteceu no final da década, quando também houve a popularização da banda larga para Internet. Dispondo de conexões rápidas o suficiente para transmitir os dados necessários, este formato cresceu vertiginosamente. Pode-se notar isto pela estrutura disponibilizada pelo primeiro servidor de

MMPG a se instalar no Brasil, ao final de 2004. Segundo Level Up! Interactive (2004, p. 3), “Suportado por uma rede atual de mais de 150 processadores, o jogo tem capacidade de conectar, ao mesmo tempo, mais de 16 mil jogadores”. Ou seja, se comparado com a estimativa feita pela IGDA, que se baseia em dados de 2002, num intervalo de aproximadamente três anos o número de usuários que um servidor precisa atender quadruplicou.

Tendo em vista os problemas de um crescimento praticamente exponencial baseado numa tecnologia com pouco tempo de pesquisa, este trabalho propõe a implementação de um protótipo de motor para servidor deste tipo de jogo, bem como a implementação de uma especificação de arquitetura para otimizar o funcionamento do mesmo.

Para a estrutura de rede, será utilizada a proposta de Lyra Studios (2003, p. 1-4), baseada em quatro camadas, incluindo uma camada cliente. Já para a implementação do software motor do servidor, será utilizada a proposta de Fiedler, Wallner e Weber (2002, p. 2-5), que descreve o modelo de publicador-assinante em canais de comunicação baseados no particionamento dos mapas de jogos, juntamente com a utilização de *Input/Output Completion Port* (IOCP) (TREGLIA, 2002, p. 506-533) para a comunicação propriamente dita entre as camadas.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um protótipo de motor de servidor de MMPG de forma genérica, que possa ser utilizado por virtualmente qualquer implementação de jogo que siga o formato de MMPG e RPG, utilizando para tal as arquiteturas descritas na seção anterior, juntamente com o interpretador e linguagem de *scripts* Lua (IERUSALIMSCHY; FIGUEIREDO; CELES, 1996).

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado da seguinte forma: o primeiro capítulo apresenta uma introdução sobre o assunto e os seus objetivos, com intuito de fornecer as informações

necessárias para seu entendimento. O segundo capítulo descreve os conceitos e técnicas que fazem parte da fundamentação teórica deste trabalho. O terceiro capítulo trata da implementação do protótipo, detalhando a especificação e desenvolvimento do mesmo. O quarto capítulo traz as conclusões, bem como sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conceitos, técnicas e trabalhos correlatos relacionados ao projeto desenvolvido.

2.1 JOGOS *ONLINE* EM MASSA E RPG

Nesta seção são vistas as definições e características de jogos *online* em massa, juntamente com as características dos jogos RPG, tanto em tabuleiros, quanto eletrônicos e no ambiente *online*. Através da união das características dos MMPG com RPG é possível definir como um servidor de jogos deste estilo deve se comportar, e quais funcionalidades deve disponibilizar.

2.1.1 Jogos *online* em massa

Jogos *online* em massa são assim chamados por reunir em uma única partida virtual um número bastante elevado de jogadores. Esta é a característica mais marcante destes jogos, e que os distingue dos jogos *multiplayer* tradicionais. Mas há também outras características que estes jogos normalmente têm, como a existência de uma espécie de mundo virtual persistente, ou seja, que continua a existir e funcionar mesmo quando um jogador não está conectado, e o estímulo à interatividade entre os próprios jogadores, através de criação de grupos, etc.

Existem no mercado os mais diversos estilos de jogos *online* em massa, entretanto, os mais comuns são os que seguem o estilo RPG.

2.1.2 RPG

Os jogos RPG não são necessariamente eletrônicos. Na verdade, começaram como jogos de tabuleiro, mas adaptaram-se muito bem ao mundo virtual. Os primeiros títulos

surgiram como jogos *offline*, que só podiam ser jogados em máquinas locais. No entanto, devido à natureza destes jogos, adaptaram-se melhor ainda ao ambiente *online* em massa.

Isso se dá devido às características de jogos RPG. Como o próprio nome diz, são jogos de interpretação de papéis. Isso quer dizer que, ao iniciar o jogo, o jogador assume o papel de um personagem com características bem definidas. Este personagem, por sua vez, é colocado em uma história, ou enredo, e trata de se desenvolver, ganhando novas habilidades através das aventuras pelas quais passa.

Nos jogos de tabuleiro, normalmente existem os jogadores e seus respectivos personagens, e há também um jogador especial chamado de mestre. Este jogador é responsável por desenvolver a história em que os outros jogadores serão inseridos, e onde deverão interpretar seus personagens.

Uma partida RPG consiste na definição de uma história por parte do jogador mestre, que passa a se tornar o mundo imaginário onde a partida vai acontecer. Os outros jogadores então criam seus personagens baseados nesse mundo definido pelo mestre. Dependendo do tipo de jogo, existem determinadas regras que tanto o mestre quanto os jogadores devem seguir. A estas regras dá-se o nome de sistema. Estes sistemas normalmente incluem perfis para cada tipo de personagem que os jogadores podem criar, bem como habilidades que cada tipo pode ter. Por exemplo, um determinado tipo de personagem pode ter uma melhor lábia que o restante, e, conseqüentemente, se sair melhor em situações em que precisa convencer outros personagens.

Com o mundo imaginário bem definido e os personagens dos jogadores criados, pode-se iniciar uma partida. Ela consiste na adaptação dos jogadores às situações que o mestre define. Ele, que não tem personagem na partida, cria situações desafiadoras através de uma narração, e os jogadores precisam adaptar seus personagens à situação, de maneira a resolver os problemas propostos pelo mestre. Desta forma, normalmente não existe a figura de ganhador ao fim de uma partida, até por que a maior parte das situações criadas pelo mestre exige que os jogadores da partida atuem em grupo. Mas é comum que os personagens dos jogadores ganhem pontos de experiência quando terminam uma partida, que podem então ser distribuídos nas habilidades de seu personagem, tornando-o mais forte. Por isso também é comum que os mesmos personagens sejam utilizados em diversas partidas, para torná-los cada vez melhores (RPG, 2006).

2.1.3 RPG *online* em massa

Para criar um *Massive Multiplayer Online Role Playng Game* (MMORPG), deve-se ter em mente o funcionamento dos RPG em tabuleiro. Isso implica na definição de uma história, que é o enredo do jogo, além da definição das classes de personagens que poderão ser utilizadas, bem como as aventuras que estes personagens poderão enfrentar.

Como a definição do jogo implica na definição de seu enredo, a figura do mestre é descartada nos MMORPGs. No entanto, todos os outros aspectos devem ser levados em consideração.

É necessário que se defina classes distintas conforme seu enredo, além de habilidades que possam ser adquiridas pelos personagens conforme eles se desenvolvem. Além disso, é preciso que sejam disponibilizadas formas para que os personagens adquiram experiência a fim de evoluir seus personagens. Na maior parte dos jogos, isso se dá através de aventuras que os personagens devem enfrentar, ou através de lutas com monstros do enredo do jogo ou mesmo com outros personagens.

No entanto, é necessário atentar ao balanço do jogo quando da definição das classes e seus atributos. Classes com habilidades mais poderosas que outras podem tornar-se um problema devido à necessidade de interação entre os jogadores. Segundo Thor (2005, p. 35, tradução nossa), "Como jogadores interagem uns com os outros, questões de balanço tornam-se muito mais vitais para o *designer* quando comparados com jogos tradicionais *single-player*". Portanto, é necessário que as classes sejam bem distintas para agradar o maior número de jogadores. Entretanto, essa diferença não pode configurar vantagens para uma classe e desvantagens para outras.

Outra característica comum em MMORPGs é a existência de equipamentos que os personagens possam utilizar, como armas, armaduras, ou qualquer outro tipo de item que os ajude a enfrentar as situações definidas pelo jogo. A forma que os personagens podem adquirir estes equipamentos varia muito conforme o jogo, mas normalmente são ganhos da mesma maneira que a experiência: através de aventuras ou de lutas.

Praticamente todas estas características podem ser vistas na figura 1, que é um *screen-shot* de um MMORPG conhecido. Nas caixas no topo à esquerda, é possível visualizar as informações do personagem, como seu nome, classe e nível. Abaixo há as informações de atributos dele, como força, agilidade, destreza, etc., além de seus equipamentos. Na caixa da direita estão as habilidades disponíveis para a classe do personagem, que foram adquiridas

conforme o personagem evoluiu.

Figura 1 – *Screen-shot* do MMORPG Ragnarök Online

Juntamente com estes equipamentos, muitos jogos disponibilizam também todo um sistema de mercado, com moeda local e possibilidade de compra e venda de itens. Com isso, forma-se uma economia semelhante à do mundo real no mundo virtual do jogo, onde personagens podem comprar e vender itens e equipamentos e fazer fortuna. Esta economia pode inclusive demonstrar características comuns à economia do mundo real, como apresentar inflação do valor dos itens quando começam a tornarem-se escassos, ou deflação quando começam a aparecer com freqüência no mercado (THOR, 2005).

2.2 ESTRUTURA DE REDE PARA JOGOS *ONLINE* EM MASSA

Devido aos requisitos de hardware, rede e disponibilidade, torna-se inviável tratar um servidor para jogos *Massive MultiPlayer* (MMP) em apenas um computador. A velocidade que os jogos crescem, tanto em tamanho e número de usuários quanto em complexidade, é superior à velocidade que cresce o desempenho do hardware disponível em um único

computador. Por isso, torna-se interessante dividir o mundo virtual em diversas máquinas no servidor.

Além das questões de desempenho, há também o fator disponibilidade: uma única máquina implica num único ponto de falha para o aplicativo, e pontos únicos de falha devem ser evitados a todo custo em aplicações deste tipo (THOR, 2005).

Estas situações geram a necessidade de separar o servidor em partes que possam ser distribuídas entre diversas máquinas. Estas máquinas e seus respectivos processos também devem estar organizados em uma arquitetura que evite os problemas descritos acima, além de estar de acordo com o *design* do jogo em questão.

Dentre os formatos utilizados, destacam-se os que separam a aplicação do servidor em camadas, conforme suas responsabilidades, de maneira que cada camada possa ser executada numa máquina distinta, e que determinadas camadas possam ser subdividas conforme a necessidade.

Lyra Studios (2003, p. 1-4), em seu produto comercial, descreve uma arquitetura para tal, composta de quatro camadas: cliente, de sessão, servidor de localização e dados. Há ainda mais uma camada alternativa para atualização do cliente (figura 2).

Fonte: adaptado de Lyra Studios (2003, p. 2).

Figura 2 – Estrutura de rede de Lyra Studios

A camada alternativa é utilizada para que a camada cliente, antes de conectar-se ao servidor, verifique se está com seu software devidamente atualizado.

A camada cliente é uma biblioteca utilizada para abstrair da implementação do jogo os detalhes de conexão e troca de mensagens com o servidor.

A camada de sessão é a camada responsável pela autenticação do usuário no momento em que se conecta ao servidor, bem como gerenciar os dados que trafegam entre a camada cliente e o servidor. Ela é necessária, pois cada mapa do jogo corresponde a uma instância do servidor de localização (quarta camada). Dessa forma, ao conectar-se com o servidor, a camada cliente na verdade conecta-se com o servidor de sessão, que por sua vez efetua a conexão com a instância do servidor de localização desejado pelo cliente, evitando que o

cliente precise conhecer os endereços dos servidores de localização. Além disso, no caso de uma mudança de mapa por parte do cliente, somente a conexão da camada de sessão com o servidor de localização é refeita, evitando várias desconexões e conexões do cliente, que poderiam tomar muito tempo. Há também a vantagem da camada de sessão poder ser distribuída entre várias máquinas separadas geograficamente, para melhorar a velocidade de conexão dos clientes de uma determinada região, garantindo a disponibilidade ao evitar um ponto único de falhas, e a escalabilidade do sistema quanto ao número de conexões suportadas. Esta camada também é responsável por carregar os dados dos usuários e seus personagens, bem como persistí-los.

A camada do servidor de localização, comumente chamado apenas de servidor, é o motor propriamente dito. Ela é responsável por manter o ambiente de um mapa do jogo. Isso implica carregar os dados do mapa, manter as informações sobre os personagens que estão neste mapa, disparar os eventos do ambiente, suportar as mensagens enviadas pelos clientes, executar os *scripts* criados pela implementação do jogo, suportar conversas entre personagens próximos, etc. Neste seu produto, Lyra Studios disponibiliza uma linguagem de *scripts* proprietária para extensão de jogos.

Já a camada de dados, como o próprio nome sugere, é o banco de dados onde os dados são persistidos. Lyra Studios utiliza o banco de dados MySQL em seu produto.

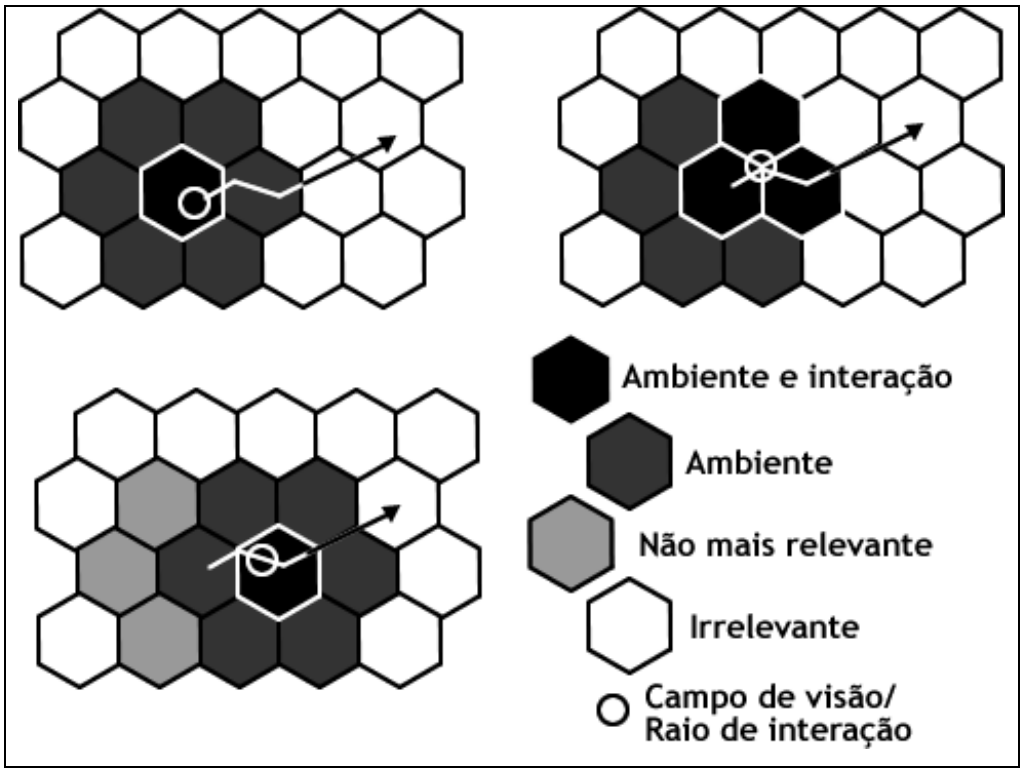
2.3 ARQUITETURA INTERNA DO SERVIDOR

Considerando o volume de informações que o servidor precisa gerenciar, é necessário definir uma arquitetura interna adequada para o software do servidor. Esta arquitetura deve se concentrar no gerenciamento dos servidores de localização (terceira camada), de maneira a facilitar o gerenciamento das informações recebidas, bem como reduzir ao máximo o volume de informações que são enviadas para os clientes, com o menor processamento possível.

Fiedler, Wallner e Weber (2002, p. 2-5) propõem uma arquitetura baseada no sistema de publicador-assinante para resolver o problema da utilização da banda de comunicação inerente a aplicações do estilo MMPG. Esta arquitetura utiliza-se da divisão do mapa do jogo em várias partes, e a criação de dois canais de comunicação para cada uma destas partes. Com a divisão do mapa, é possível restringir o volume de dados a serem enviados para os clientes a apenas à parte em que o seu personagem se encontra, e as partes adjacentes a esta e que estão

no campo de visão do jogador. Já a criação dos dois canais tem por objetivo enviar apenas os dados realmente necessários de cada parte do mapa para o usuário.

Os canais de transmissão são o de ambiente e o de interação. O canal de ambiente transmite informações sobre o ambiente de uma parte do mapa, ou seja, seu estado, os personagens, objetos, acontecimentos, etc. O canal de interação, por sua vez, transmite informações referentes à interação do personagem com outros personagens e ambiente ao seu redor, como conversas, lutas, etc. Como um personagem só pode interagir com coisas realmente próximas a ele, é necessário a transmissão dos dados de interação apenas da parte do mapa onde o personagem se encontra, ou, no máximo, algumas das partes adjacentes quando o personagem se encontrar na divisão destas partes.



Fonte: adaptado de Fiedler, Wallner e Weber (2002, p. 4).

Figura 3 – Visão geral do formato de publicador/assinante e seus canais de comunicação

A escolha de hexágonos para a divisão se dá com o intuito de diminuir ao máximo o número de partes adjacentes a cada parte do mapa. No entanto, segundo Fiedler, Wallner e Weber (2002, p. 6), “gerenciar partes retangulares é muito simples, em comparação com partes hexagonais”. Desta forma, quando se trata da implementação do modelo, a escolha de partes retangulares ou quadradas representa vantagem sobre os hexágonos, devido aos ganhos de desempenho e simplificação do código.

Da mesma forma que na seção anterior, no modelo de publicador/assinante levanta-se a necessidade de separar o servidor em diversas máquinas, para que ele seja capaz de

responder ao volume de conexões e processamento necessários. No entanto, neste modelo a separação se dá apenas com o gerenciamento de partes do mapa em máquinas separadas. Ou seja, o modelo permite que as partes em que um único mapa foi dividido sejam distribuídas por diversas máquinas numa rede. Desta maneira a escalabilidade do servidor torna-se limitada virtualmente apenas pelo volume de dados que os clientes podem receber, pois o processamento no lado do servidor pode ser dividido em quantas máquinas forem necessárias, apenas ajustando-se o tamanho das partes dos mapas.

Levando isto em conta, são abordados aspectos de implementação que permitam a utilização do modelo num ambiente de *cluster* no lado do servidor. Entretanto, mesmo o servidor sendo composto de diversas máquinas, o modelo mantém isso transparente ao cliente, de forma que ele visualize apenas um grande mapa, onde na verdade existem diversas partes separadas em máquinas diferentes, que podem inclusive estar separadas geograficamente e conectadas através da Internet.

Para atingir este objetivo, o modelo define o protocolo de comunicação a ser utilizado na rede do servidor, bem como a arquitetura do software que fará o gerenciamento dos canais de comunicação, sejam de ambiente ou de interação.

2.3.1 Protocolos de comunicação

Considera-se que a implementação do modelo publicador/assinante utiliza a Internet para a comunicação com máquinas fora da rede local e deve manter um comportamento similar quando executando numa rede local. Desta forma, os protocolos de rede que são viáveis para a utilização na comunicação entre os canais, ficam resumidos aos do *Internet Protocol* (IP) (FIEDLER; WALLNER; WEBER, 2002):

a) UPD *unicast*: é um protocolo de datagrama pouco confiável não orientado a conexões. Não é confiável pois não garante que a mensagem será recebida corretamente pelo destinatário. Por se tratar de um protocolo *unicast*, pode ser enviado apenas para um cliente de cada vez;
b) *Transmission Control Protocol* (TCP) *unicast*: é um protocolo de datagrama confiável, orientado a conexões. Ele conta com diversas técnicas que visam garantir que o destinatário receba as mensagens corretamente. No entanto, por ser *unicast*, também é limitado a um único cliente por vez;
c) *User Datagram Protocol* (UDP) *broadcast*: é similar ao UDP *unicast*, sendo que

única diferença é que suporta o envio de mensagens a todos os nós da sub-rede de quem envia a mensagem. Entretanto, este protocolo não costuma ser roteado através da Internet.

Para a comunicação entre os canais de ambiente, o protocolo utilizado é o TCP *unicast*, pois se tratam de canais que geram um volume menor de tráfego, onde as mensagens podem até ser atrasadas, no entanto, não podem ser perdidas.

Para os canais de interação, são utilizados dois protocolos, conforme a situação. Quando a comunicação se dá numa rede local, é utilizado o protocolo UDP *broadcast*, devido ao ganho de desempenho em enviar uma única vez as mensagens a todos os destinatários. Já para comunicações com partes fora da rede local, onde é necessário enviar as mensagens através da Internet, é utilizado o protocolo TCP *unicast*, devido à sua confiabilidade (FIEDLER; WALLNER; WEBER, 2002).

2.3.2 Gerenciamento dos canais de comunicação

O gerenciamento dos canais de comunicação leva em consideração a possibilidade das partes dos mapas estarem não só em máquinas separadas, como separadas pela internet. E, mesmo assim, a conexão entre eles deve ser transparente: um cliente assina um canal, sem precisar se preocupar com a localização do gerenciador deste canal ou da máquina que faz o gerenciamento da parte do mapa ao qual o canal está ligado.

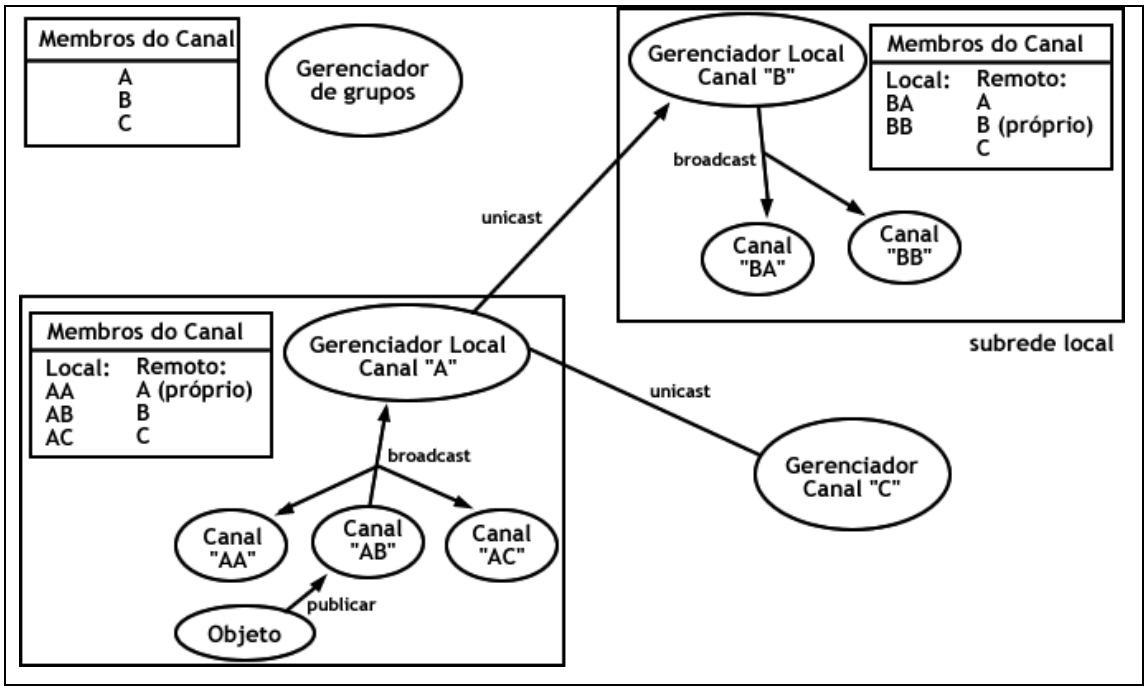
Para garantir estas características, são utilizados dois tipos de gerenciadores: gerenciadores locais e gerenciadores de grupo.

Para cada rede local, existe um gerenciador local de canais, que fica responsável por gerenciar os canais das partes de mapas distribuídas nas máquinas de sua rede. Ele também fica responsável por fazer o encaminhamento correto de solicitações a canais que não estão sob sua responsabilidade. Para isto, este gerenciador pode conectar-se com outros gerenciadores em redes remotas, fazendo a ponte entre os publicadores/assinantes de sua rede local com os publicadores/assinantes da rede remota.

O gerenciador de grupo, por sua vez, fica responsável por manter as informações dos gerenciadores de cada canal, pois é através dele que os gerenciadores locais descobrem com qual gerenciador remoto devem efetuar a conexão ao receber solicitações para canais que não estão sob sua responsabilidade.

Como pode ser visto na figura 4, quando um objeto envia uma mensagem a um canal,

ele é primeiramente distribuído aos outros objetos e canais da rede local através de uma mensagem que, dependendo do tipo de canal, pode ser *broadcast*. A mensagem também é enviada ao gerenciador local da rede, que, por sua vez, envia aos gerenciadores remotos necessários. Desta forma, o processo fica transparente para o objeto. Ao requisitar um canal que está fora da rede local, a requisição é feita, na verdade, ao gerenciador local de conexões. Através do gerenciador de grupos, o gerenciador local descobre qual o gerenciador remoto responsável pelo canal desejado, efetua a conexão com este gerenciador e passa a enviar as mensagens até lá.



Fonte: adaptado de Fiedler, Wallner e Weber (2002, p. 7).

Figura 4 – Visão geral do gerenciamento de canais em redes locais e remotas

2.4 GERENCIAMENTO DE CONEXÕES

Um dos maiores desafios na construção de um servidor eficiente para MMPGs é o gerenciamento eficiente das conexões com os clientes. O servidor deve ser capaz de suportar um número elevado de conexões, e ainda assim não comprometer o desempenho da máquina, pois precisa efetuar o processamento das mensagens enviadas pelos clientes no contexto do jogo, e responder a elas em tempo real. Nesta seção serão apresentadas as principais técnicas de gerenciamento de conexões em aplicações de alto *throughput*.

2.4.1 *Sockets* assíncronos

Os *sockets* assíncronos (*non-blocking*) estão entre as primeiras técnicas para construção de aplicações que precisam gerenciar várias conexões. Estes *sockets* têm a vantagem de não bloquearem a aplicação durante operações de escrita ou leitura. Nestes casos, o aplicativo não precisa esperar pelo término das operações. Após a chamada de uma função de escrita/leitura, a execução do programa passa diretamente para as próximas instruções (SIKORA, 2001). Além disso, estes tipos de *sockets* são suportados em diversas plataformas, tornando a aplicação facilmente portável (TREGLIA, 2002).

A utilização desta estratégia consiste na criação de *sockets* para as conexões recebidas no servidor. Estes *sockets* são então armazenados em estruturas do tipo *FD_SET*, e através destas estruturas a aplicação faz verificações de tempos em tempos em busca de *sockets* que possuam operações pendentes.

Por padrão, esta verificação é feita através da função *select()*. Esta função retorna o número de *sockets* com operações pendentes ou que estão em algum estado anormal dentre os *sockets* de estruturas *FD_SET*. Como esta função retorna apenas se há *sockets* ativos na estrutura, torna-se necessário verificar dentro da estrutura quais *sockets* estão ativos, um a um.

```
int select(
  int nfds,                         //ignorado, existe por retro-compatibilidade
  fd_set* readfds,                  //sockets a verificar status de leitura
  fd_set* writefds,                 //sockets a verificar status de escrita
  fd_set* exceptfds,                //sockets a verificar status de erro
  const struct timeval* timeout     //timeout
);
```

Fonte: adaptado de Microsoft Corporation (2006).

Quadro 1 – Declaração da função *select()*

Este modelo funciona bem para aplicações que gerenciam algumas dezenas de conexões, mas não se aplica bem ao gerenciamento de centenas ou milhares de sessões ativas. Para isso, os sistemas operacionais mais modernos disponibilizam *Application Programing Interfaces* (APIs) estendidas que são mais eficientes que serviços básicos de *sockets* (TREGLIA, 2002).

2.4.2 *Sockets* gerenciados por mensagens do Windows

Esta estratégia é uma extensão dos *sockets* assíncronos. Há também alguma confusão

quanto a nomeação desta técnica: alguns autores a chamam de *sockets* assíncronos (e a estratégia vista no item anterior de *non-blocking*). No entanto, a forma mais comum é referir-se a ela como *sockets* gerenciados por mensagens, devido à sua natureza. O objetivo dela é eliminar a necessidade da varredura nas estruturas *FD_SET* em busca de *sockets* ativos. Nestes casos, após criar um *socket non-blocking* para uma nova conexão, deve-se invocar a API `WSAAsyncSelect()` do Windows (quadro 2).

```
int WSAAsyncSelect(
  SOCKET s,           //descritor do socket a associar
  HWND hWnd,          //handle da janela que receberá as mensagens
  unsigned int wMsg, //mensagem enviada à janela em eventos
  long lEvent         //eventos que a aplicação deseja receber
);
```

Fonte: adaptado de Microsoft Corporation (2006).

Quadro 2 – Declaração da função *WSAAsyncSelect()*

Com isso, o *socket* passa a ser gerenciado pelo sistema operacional, que se utiliza do código de mensagem do terceiro parâmetro para notificar a aplicação quando operações são completadas. A aplicação pode ainda, através do último parâmetro, configurar para quais tipos de operações deseja ser notificado.

Vale ressaltar que quando se fala em mensagens, neste caso, refere-se às mensagens que o Windows utiliza para notificar aplicações de eventos, e não a mensagens enviadas por rede, por exemplo. Este é o padrão que o Windows utiliza para notificar as aplicações de eventos diversos, como cliques de mouse, teclas pressionadas, etc.

Dentre os dados que a aplicação recebe de uma mensagem que o sistema operacional envia, está o *socket* que teve a operação completada. Desta maneira, a aplicação pode responder diretamente a ele, sem precisar iterar por listas. No exemplo do quadro 3, pode-se perceber isso. São recebidos três valores: o código da mensagem (`message`), a operação que disparou o evento (`lParam`) e o *socket* em que o evento ocorreu (`wParam`).

Entretanto, como a notificação por parte do sistema operacional vem no formato de mensagens, é necessário que a aplicação seja construída no formato de janela. Aplicações do tipo console, por exemplo, não possuem janelas e consequentemente não podem receber as mensagens do Windows.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
    ...
    //verifica qual o tipo de mensagem
    witch (message) {
        //mensagem de notificação de socket
        case WM_WSAASYNC:
        {
            //verifica a operação
            switch (WSAGETSELECTEVENT(lParam))
            {
                //operação de conexão
                case FD_ACCEPT:
                {
                    //aceita a conexão recebida (wParam contém o socket
                    //que disparou a notificação)
                    sAccept = accept(wParam,NULL,NULL);
                    ...
                }
                //operacao de leitura
                case FD_READ:
                {
                    ...
                }
                //operação de escrita
                case FD_WRITE:
                {
                    ...
                }
                //finalização da conexão
                case FD_CLOSE:
                {
                    ...
                }
            }
        }
    }
    return 0;
}
```

Fonte: adaptado de Sikora (2001).

Quadro 3 – Exemplo de rotina de recebimento de notificações de *sockets*

2.4.3 IOCP

Das técnicas de gerenciamento de conexão, a utilização de IOCP destaca-se em aplicações que, além de gerenciar um volume muito grande de dados e conexões, necessitem também de alta escalabilidade. Mecanismos como as funções *WSAAsyncSelect* e *select* não fazem uso das possibilidades que o sistema operacional oferece, pois foram criadas para facilitar a portabilidade de aplicações desenvolvidas para Windows 3.1 e UNIX, respectivamente (JONES; DESHPANDE, 1999). Ela também se utiliza de *sockets non-blocking*, porém, possui um gerenciamento mais avançado das notificações, permitindo, entre

outras coisas, que a aplicação trabalhe com diversos *threads* para verificar o status dos *sockets* e processamento das mensagens enviadas e recebidas em uma única *Completion Port* (CP). Além disso, não há a necessidade de criação de janelas como no caso dos *sockets* gerenciados por mensagens do Windows, e o acesso às conexões com eventos pendentes se dá sem a necessidade de iteração em listas, como é o caso dos *sockets* assíncronos.

Segundo Jones e Deshpande (1999) “Uma CP é uma fila na qual o sistema operacional coloca notificações de operações assíncronas completadas. Uma vez terminada uma operação, uma notificação é enviada para um *thread* de trabalho, que pode processar o resultado”.

Para utilizar este esquema, deve-se criar uma CP através das APIs disponibilizadas pelo Windows. Após isso, os *sockets* criados para as conexões recebidas devem ser associados a ela, para que o sistema operacional possa passar a gerenciar suas operações de leitura e escrita.

No momento da associação do *socket* à CP, é possível também passar um ponteiro com informações específicas do *socket*. Este ponteiro pode ser utilizado, por exemplo, para manter a ligação entre o *socket* e os dados do cliente que está conectado.

```
void criaIOCP()
{
    //cria o handler da completion port
    HANDLE hIocp;

    //cria a completion port propriamente dita
    hIocp = CreateIoCompletionPort(
        INVALID_HANDLE_VALUE,
        NULL,
        (ULONG_PTR)0,
        0);
    if (hIocp == NULL) {
        //tratamento de erro...
    }
    ...
    //cria um socket
    SOCKET s;
    s = socket(AF_INET, SOCK_STREAM, 0);

    if (s == INVALID_SOCKET) {
        //tratamento de erro...
    }
    //Associa o socket à completion port criada anteriormente
    if (CreateIoCompletionPort((HANDLE)s,
        hIocp, //completion port para associar ao socket
        (ULONG_PTR)0, //ponteiro para informações do socket
        0) == NULL) {
            //tratamento de erro...
    }
    ...
}
```

Fonte: adaptado de Jones e Deshpande (1999).

Quadro 4 – Exemplo de criação de CP e associação a um socket

Com a CP criada e os *sockets* associados, inicia-se o processamento da conexão propriamente dita, que deve ser feita por um *thread* de trabalho. Este *thread* fica responsável por verificar de tempos em tempos o estado das conexões associadas a uma determinada CP. Esta verificação normalmente se dá através de um laço numa rotina que invoque a função `GetQueuedCompletionStatus`. Esta função, para cada passagem, retorna as notificações de conclusão de operações dos *sockets* associados à CP.

```
BOOL GetQueuedCompletionStatus(
  HANDLE CompletionPort,
  LPDWORD lpNumberOfBytes,
  PULONG_PTR lpCompletionKey,
  LPOVERLAPPED* lpOverlapped,
  DWORD dwMilliseconds
);
```

Fonte: Microsoft Corporation (2006).

Quadro 5 – Declaração da função *GetQueuedCompletionStatus*

Os *threads* de trabalho, além de chamar a função, também ficam responsáveis por copiar as informações recebidas da operação para os *buffers* da aplicação, que são então processados assincronamente por outros *threads* que possuem a lógica da aplicação propriamente dita.

Estas notificações vêm acompanhadas de um ponteiro para uma estrutura *overlapped*. Estas estruturas são as mesmas que podem ser passadas para as funções assíncronas de leitura ou escrita que originaram a notificação, e são retornadas para que a aplicação possa manter a rastreabilidade das operações completadas. No entanto, as estruturas *overlapped* por si só não possuem as informações necessárias para a aplicação identificar qual operação foi completada. Por isso, é recomendado utilizar uma estrutura própria, que além dos campos originais, possua campos que a aplicação possa utilizar para identificar e rastrear as operações (JONES; DESHPANDE, 1999).

Embora esta estratégia apresente diversas vantagens, há alguns cuidados que devem ser tomados durante sua implementação. O principal deles é o sincronismo entre os *threads* que farão o trabalho de leitura/escrita dos dados, e os *threads* que farão o processamento da lógica da aplicação. Como é necessário que os *threads* de trabalho passem aos *threads* da lógica da aplicação as mensagens recebidas e/ou enviadas para que elas sejam processadas, é necessário que o acesso a estes *buffers* esteja devidamente sincronizado para evitar problemas.

Além disso, é necessário fazer um bom gerenciamento dos *buffers* de leitura e escrita do sistema operacional. Devido à arquitetura do Windows, existem *buffers* internos no sistema operacional para armazenar temporariamente as informações de leitura ou escrita em *sockets* até que a aplicação os solicite. Ou seja, ao efetuar uma operação de escrita, os dados a

ser enviados são primeiramente copiados para um *buffer* interno do sistema operacional, para depois serem efetivamente enviados pelas camadas de transporte do núcleo do sistema. Estes *buffers*, por serem considerados temporários, não são paginados, ficando limitados ao volume de memória RAM que o sistema operacional reserva para operações não paginadas.

Por isso, para evitar que aconteça uma sobrecarga nestes *buffers*, a aplicação deve gerenciá-los corretamente, sempre evitando que informações fiquem acumuladas por muito tempo.

Há também a possibilidade de desabilitar os *buffers* do sistema operacional, para evitar a necessidade da cópia extra dos *buffers* da aplicação para estes *buffers*. No entanto, segundo Jones e Deshpande (1999), "desligar os *buffers* de recebimento [...] não oferece ganho real de desempenho". Isso se dá, por que mesmo nestes casos esta cópia de buffers acaba sendo efetuada em níveis abaixo do Winsock (camada do sistema operacional responsável pelo gerenciamento de *sockets*). Além disso, ao fazer isso, os *buffers* da aplicação acabam sendo bloqueados pelo núcleo do sistema operacional, e só são desbloqueados quando a operação é finalizada. Isso leva ao mesmo problema de limitação de recursos visto no parágrafo anterior.

2.5 LINGUAGENS DE *SCRIPTS* PARA EXTENSÃO

Jogos no estilo MMP baseiam-se num gigante mundo virtual, que, assim como o mundo real, deve ser dinâmico. Além disso, é comum que um jogador necessite de um tempo considerável de jogo para evoluir seus personagens aos mais altos níveis. Por isso, é importante que o servidor de um MMPG seja flexível o bastante para que alterações no enredo e mesmo na mecânica do jogo possam ser feitas facilmente, de maneira a trazer novidades freqüentes aos jogadores, mantendo-os ativos.

Uma das saídas mais comuns para tornar o motor do servidor flexível para estas alterações constantes, sem que isso se reflita num aumento excessivo de custos, é a utilização de linguagens de *scripts* para extensão. Segundo Thor (2005, p. 199) "um sistema de *scripts* sólido provou-se a única opção viável para suportar o conteúdo dinâmico de um título MMPG".

É comum entre os desenvolvedores a utilização de linguagens de *script* de propósito geral já existentes, como Phyton, Lua ou LISP (THOR, 2005). Estas linguagens, além de serem, em geral, gratuitas para utilização e distribuição, são completas o suficiente para

suprirem as necessidades de um motor MMPG.

Dentre estas linguagens, uma que vem se destacando na indústria de jogos é a brasileira Lua, conforme é visto a seguir.

2.5.1 Linguagem e interpretador Lua

Segundo Ierusalimschy, Figueiredo e Celes (1996), Lua é “uma linguagem procedural extensível, com poderosos facilitadores para descrição de dados, desenvolvida para ser utilizada como uma linguagem de extensão para propósitos gerais”. Ela foi criada exclusivamente para extensão de aplicações, portanto, não pode executar independentemente; é sempre necessário existir uma aplicação hospedeira que execute os *scripts* escritos nesta linguagem, através de bibliotecas que são disponibilizadas para compilá-los e interpretá-los.

Atualmente encontra-se na versão 5.11, e é implementada como uma biblioteca utilizando ANSI C. Os *scripts* escritos na linguagem são pré-compilados para um *bytecode* interno, de maneira a melhorar o desempenho de sua execução. Ela própria também conta com um *garbage collector*, que faz o gerenciamento automático da memória utilizada pelos *scripts*, evitando que a aplicação precise preocupar-se com isso.

Ela é uma linguagem *case-sensitive*, composta das palavras reservadas e *tokens* que podem ser vistos no quadro 6.

and	break	do	else	elseif		
end	false	for	function	if		
in	local	nil	not	or		
repeat	return	then	true	until	while	
+	-	*	/	%	^	#
==	~=	<=	>=	<	>	=
(	)	{	}	[	]	
;	:	,	.	..	...	

Fonte: adaptado de Ierusalimschy, Figueiredo e Celes (2006).

Quadro 6 – Palavras reservadas e *tokens* da linguagem Lua

Como em qualquer outra linguagem, estas palavras e *tokens* não podem ser utilizadas em identificadores. Nos próximos capítulos são descritos em maiores detalhes sua sintaxe, principais funcionalidades e forma de integração com aplicações hospedeiras.

2.5.1.1 Sintaxe

Como Lua executa apenas dentro de uma aplicação hospedeira, não existe um "programa principal", que possa ser executado independentemente. Os *scripts* Lua são apenas seqüências de comandos, que são executados por uma aplicação qualquer. Estas seqüências de comandos, ou *scripts*, são chamadas de *chunks*.

A sintaxe da linguagem Lua, embora seja bastante parecida com Pascal, tem algumas características da linguagem C também. Ela suporta praticamente todas as principais construções destas duas linguagens, como atribuições, estruturas de controle, declaração de funções, variáveis, etc.

As estruturas de controle de fluxo disponíveis na linguagem são: *if*, *while*, *repeat* e *for*. Estas estruturas suportam blocos de comandos, no entanto, não é necessária a utilização dos marcadores de blocos *begin* e *end*, como é feito na linguagem Pascal. Nestes casos, é utilizada apenas a palavra reservada *end* para finalizar o bloco (exceto no caso do comando *repeat*, pois ele é auto-contido). A palavra *end* deve ser utilizada mesmo quando o bloco é composto de apenas um comando, o que não é necessário no Pascal.

```
stat ::= while exp do block end
stat ::= repeat block until exp
stat ::= if exp then block {elseif exp then block} [else block] end
```

Fonte: Ierusalimschy, Figueiredo e Celes (2006).

Quadro 7 – *Backus Naur Form* (BNF) das estruturas de controle de fluxo da linguagem Lua

Os comandos de atribuição em Lua se dão através do símbolo igual "=". Por conseqüência o símbolo de comparação é "==", seguindo o padrão da linguagem C. Lua suporta ainda atribuições múltiplas, ou seja, atribuir de uma única vez uma lista de valores a uma lista de variáveis, desde que as listas sejam de tamanhos iguais. Embora seja possível efetuar isto com listas de tamanhos diferentes, apenas as variáveis que possuem valores correspondentes serão atribuídas. No caso da lista de valores ser menor que a lista de variáveis, as variáveis que não têm valor correspondente são preenchidas com *nil*, que é a representação de nulo da linguagem. Já no caso da lista de valores ser maior que a lista de variáveis, os valores excedentes são simplesmente ignorados. Além disso, expressões complexas também são suportadas em atribuições.

```
--troca os valores entre as variáveis a e b
a, b = b, a
--apenas as variaveis v1 e v2 receberão os valores...
--v3 receberá nil.
v1, v2, v3 = "Teste1", "Teste2"
--o valor 10 será descartado na atribuição...
v4, v5 = 1, 2, 10
```

Quadro 8 – Atribuições na linguagem Lua

Na linguagem Lua existem três tipos diferentes de variáveis: globais, locais e *arrays* associativos dinâmicos (ou tabelas). A declaração de variáveis pode acontecer em qualquer lugar de um *chunck*. Para declarar uma variável, basta atribuir algum valor a um identificador. A partir do momento que isto é feito, a variável é criada.

O escopo de uma variável local em Lua é semelhante ao da linguagem C. Segundo Ierusalimschy, Figueiredo e Celes (2006, tradução nossa), "O escopo de uma variável começa no primeiro comando após sua declaração, e dura até o término do bloco que inclui a declaração". É possível também utilizar a palavra reservada *local* para definir variáveis locais a um determinado escopo, mesmo que com nome igual a variáveis já declaradas fora de seu escopo.

Na linguagem Lua, as variáveis não são tipadas, apenas seus valores. Desta forma, é possível atribuir qualquer tipo de dado a qualquer variável. São suportados oito tipos de valores para as variáveis (IERUSALIMSCHY; FIGUEIREDO; CELES, 2006):

a) *nil*: indica um valor nulo, que não existe;
b) *boolean*: representa um valor booleano, que pode assumir os valores verdadeiro (*true*) ou falso (*false*);
c) *number*: representa um valor numérico de ponto flutuante;
d) *string*: representa um valor alfanumérico;
e) *function*: representa um ponteiro para uma função;
f) *userdata*: representa valores específicos da linguagem C, armazenados numa estrutura específica da linguagem Lua;
g) *thread*: representa um *thread* independente, utilizado no esquema de co-rotinas disponibilizado pela linguagem, e não deve ser confundido com *threads* do sistema operacional;
h) *table*: indica um *array* associativo dinâmico, que é visto em detalhes no próximo capítulo.

A avaliação dos tipos dos dados das variáveis durante a realização de expressões é feita e tratada automaticamente pela linguagem. Apenas quando há tipos realmente

incompatíveis acontecem os erros, que podem ser tratados pelo usuário através das funções de *fallback*, que são vistas na seção 2.5.1.4.

A linguagem Lua também é conhecida por ser extensível, pois, em seu *design*, conta com diversos meta-mecanismos que permitem aos próprios programadores desenvolver novas características. Estes meta-mecanismos são: *arrays* associativos dinâmicos, facilitadores de reflexão e *fallbacks* (IERUSALIMSCHY; FIGUEIREDO; CELES, 1996).

2.5.1.2 Arrays associativos dinâmicos

São, em resumo, tabelas dinâmicas que podem ser declaradas no código Lua. Estas tabelas, além de não necessitarem ser homogêneas, podem ter seus campos acessados diretamente no formato "tabela.campo". Desta maneira, pode-se estender a sintaxe Lua conforme as tabelas definidas no *script*.

```
2
3  --Define a tabela windowl, com campos 'x', 'y' e 'caption', iniciando
4  --os mesmos com seus respectivos valores
5  windowl = {x = 200, y = 300, caption = "Teste de tabelas Lua"}
6  --utiliza os campos da tabela definida acima, acessando-os diretamente
7  ret = criaJanela(windowl.x, windol.y, windowl.caption);
8
```

Quadro 9 – Utilização de *arrays* associativos em Lua

Além disso, ao contrário de outras linguagens, estas tabelas não ficam ligadas diretamente a uma variável, mas são tratadas de forma semelhante a ponteiros. Desta maneira, várias variáveis podem apontar para uma única tabela, bem como tabelas podem referenciar outras tabelas.

```
list = {} --cria uma tabela vazia
current = list --define a primeira tabela da lista
i = 0
while i < 10 do
  current.value = i
  current.next = {} --próximo é uma tabela vazia
  current = current.next --efetua encadeamento com anterior
  i = i+1
end
current.value = i
current.next = list --torna a lista circular
```

Fonte: adaptado de Ierusalimschy, Figueiredo e Celes (1996)

Quadro 10 – Definição de tabelas encadeadas em Lua

Dentre outras funcionalidades, pode-se destacar também que os construtores das tabelas aceitam expressões, e com isso, é possível criar tabelas aninhadas.

2.5.1.3 Reflexão de código

Lua também suporta alguns facilitadores para reflexão de código. Isso se dá através das funções "*next*", que navega recursivamente entre os campos de uma tabela, e "*nextvar"*, que navega entre as variáveis declaradas num *script* Lua. Com estas funções é possível extrair os índices (campos no caso de tabelas e nome das variáveis para as variáveis) e seus respectivos valores dinamicamente (quadro 11). Desta forma, pode-se escrever rotinas genéricas para persistência ou clonagem de tabelas ou variáveis quaisquer, por exemplo.

```
--função que clona um objeto qualquer
function clone (o)
  local new_o = {}          -- cria um novo objeto
  local i, v = next(o,nil) -- pega o primeiro índice "o", junto com seu valor
  while i do
    new_o[i] = v            -- armazena eles numa nova tabela
    i, v = next(o,i)        -- pega o próximo índice e seu valor
  end
  return new_o
end
```

Fonte: adaptado de Ierusalimschy, Figueiredo e Celes (1996).

Quadro 11 – Função genérica de clonagem de objetos Lua

2.5.1.4 Fallbacks

Por se tratar de uma linguagem não tipada, Lua está sujeita a diversas situações anormais durante a execução. Por exemplo, operações aritméticas aplicadas a operadores não numéricos, tentar indexar um valor inexistente numa tabela, ou tentar invocar uma função inexistente (IERUSALIMSCHY; FIGUEIREDO; CELES, 1996).

Como parar a execução nestas situações não seria conveniente para uma linguagem embutida, Lua permite aos programadores indicar funções específicas para tratar estas situações de erro. Estas funções são chamadas de funções de *fallback*. Através da função *setfallback()*, é possível associar funções escritas pelo programador com as situações de erros descritas acima. Desta forma, quando um erro acontecer o código será desviado para a função de *fallback* associada ao erro, que deve fazer o seu tratamento.

2.5.1.5 Integração do interpretador Lua a aplicações

Como Lua não executa de maneira independente, são disponibilizadas bibliotecas para facilitar a integração de seu interpretador com aplicações que servirão de hospedeiras para seus *scripts*. Estas bibliotecas são disponibilizadas gratuitamente, permitindo inclusive a distribuição da aplicação que as utiliza, sem a necessidade de distribuir seu código-fonte, e são implementadas utilizando ANSI C.

A aplicação hospedeira pode executar os *script*s Lua carregando arquivos, ou mesmo através de *strings*, já que as APIs disponibilizadas pelas bibliotecas suportam ambos os formatos. Os *scripts*, ou *chuncks*, executam sempre num ambiente global Lua. Ou seja, mesmo que *scripts* sejam executados separadamente em momentos distintos pela aplicação, eles podem compartilhar variáveis globais. Da mesma forma, a aplicação hospedeira pode acessar estas variáveis e ler ou modificar os seus valores.

A comunicação entre os *script*s executados e a aplicação hospedeira se dá através de uma pilha virtual, onde são adicionados os parâmetros que a aplicação deseja enviar ao *script*, e o retorno do *script* ao final de sua execução. A aplicação pode acessar os valores desejados na pilha através de um índice simples, onde o índice um indica o primeiro valor adicionado à pilha, e o índice *n* indica o último valor adicionado à mesma.

Além disto, é possível disponibilizar para os *scripts* Lua funções internas da aplicação. Desta forma, pode-se criar uma *interface* da aplicação com os *scripts*, publicando suas operações internas.

```
#include <stdio.h>
#include "lua.h"    //cabeçalho principal Lua
#include "lualib.h" //bibliotecas extras (opcional)

int main (int argc, char *argv[])
{
    char linha[BUFSIZ]; //buffer do script
    lua_State* pilha;  //pilha virtual Lua

    //inicia a biblioteca de I/O (opcional)
    iolib_open();
    //inicia a biblioteca de strings (opcional)
    strlib_open();
    //inicia a biblioteca de funções matemáticas (opcional)
    mathlib_open();
    //lê o comando do teclado e executa ele
    while (gets(linha) != 0)
        luaL_dostring(pilha,linha);

    //pega o retorno na pilha, na primeira posição
    lua_Number ret = lua_tonumber(pilha,1);
}
```

Fonte: adaptado de Ierusalimschy, Figueiredo e Celes (1996).

Quadro 12 – Exemplo de um interpretador interativo de *scripts* Lua

2.6 TRABALHOS CORRELATOS

Bogojevic (2003) apresenta em sua tese uma discussão sobre a arquitetura de um jogo MMPG, envolvendo todos os seus aspectos, tanto do enredo quanto da implementação propriamente dita. Apesar de tratar dos aspectos de um MMPG por completo, ele é focado em jogos que simulam partidas de futebol. São descritos os principais problemas a serem enfrentados na implementação de uma arquitetura MMPG, incluindo técnicas de design, aspectos de rede e de segurança, implicações da utilização de computação gráfica, sistemas de bancos de dados e finalmente uma análise de mercado para este tipo de jogo.

Mesmo sendo focado em jogos de futebol, este trabalho é de grande utilidade neste projeto, especialmente na elicitação dos requisitos, pois já lista os mais importantes e que devem ser atendidos pela implementação.

Santos (1996) propõe a utilização da linguagem Lua na implementação de um *framework* para suporte a objetos visuais interativos. A implementação do *framework* se dá de forma a ser um servidor de eventos e desenho para objetos visuais descritos em Lua, desta forma garantindo a flexibilidade suficiente para que o próprio usuário consiga descrever seus objetos visuais. Este *framework* visa a utilização da linguagem Lua para os mesmos fins e da mesma forma que é necessária neste projeto, de maneira que foi de grande valia no momento da implementação da *interface* com este interpretador.

3 DESENVOLVIMENTO DO TRABALHO

O presente capítulo descreve a especificação, implementação e os testes do protótipo implementado. Na primeira seção são descritos os requisitos do aplicativo. A segunda seção mostra a especificação do projeto, através de diagramas de casos de uso, atividades, classes, entidade-relacionamento, além de descrições textuais. Na terceira seção é descrita a implementação do protótipo, técnicas e ferramentas utilizadas, além da operacionalidade da implementação através de um caso de uso. Na quarta seção estão os resultados obtidos e as discussões referentes aos trabalhos correlatos apresentados na fundamentação teórica.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Nesta seção são descritos os principais requisitos de cada camada do protótipo, classificados como funcionais (RF) ou não-funcionais (RNF).

A camada do servidor de sessão deverá:

a) gerenciar as conexões dos usuários, incluindo a autenticação deles (RF);
b) efetuar a carga e persistência dos dados dos usuários quando necessário (RF);
c) carregar as informações de mapas do mundo virtual (RF);
d) disparar eventos conforme as ações efetuadas por jogadores ou acontecimentos do ambiente do mundo virtual (RF);
e) permitir a criação e execução de *scripts* de extensão para os eventos disparados pelo motor (RF);
f) permitir ao usuário conversar com personagens próximos ao seu no mundo virtual (RF);
g) permitir ao usuário movimentar seu personagem pelo mundo virtual (RF);
h) implementar as funções necessárias para comunicação com as camadas de localização, permitindo a assinatura dos canais de interação e/ou ambiente destes servidores (RF);
i) utilizar o interpretador e linguagem de *scripts* Lua como ferramenta para execução de *scripts* de extensão (RNF);
j) utilizar banco de dados MySQL para armazenar os dados dos usuários e os dados

do ambiente do mundo virtual (RNF);

k) ser implementado na linguagem C++, utilizando o ambiente Microsoft Visual Studio .Net 2003 (RNF).

A camada de localização deverá:

a) carregar e gerenciar partes de mapas do mundo virtual, conforme a configuração de cada instância do servidor, controlando as ações e eventos que acontecem em cada uma delas (RF);
b) disponibilizar canais de comunicação de eventos de ambiente e de interação para cada uma das partes de mapas gerenciadas pelo servidor (RF);
c) permitir que servidores de sessão conectem-se e assinem os canais disponibilizados, gerenciando a atualização destes assinantes conforme acontecem os eventos de ambiente (RF);
d) disparar eventos conforme as ações efetuadas por jogadores ou acontecimentos do ambiente do mundo virtual (RF);
e) permitir a criação e execução de *scripts* de extensão para os eventos disparados pelo motor (RF);
f) utilizar o interpretador e linguagem de *scripts* Lua como ferramenta para execução de *scripts* de extensão (RNF);
g) utilizar banco de dados MySQL para armazenar os dados dos usuários e os dados do ambiente do mundo virtual (RNF);
h) ser implementado na linguagem C++, utilizando o ambiente Microsoft Visual Studio .Net 2003 (RNF).

3.2 ESPECIFICAÇÃO

Nesta seção é descrita a especificação do protótipo, através da utilização de ilustrações, descrições textuais, dos diagramas de casos de uso, atividades e classes da *Unified Modeling Language* (UML). Também é especificado seu modelo de dados, através de diagramas entidade-relacionamento. Além disso, o protocolo utilizado na comunicação entre as camadas e os eventos que devem ser disparados pelo motor também são descritos.

3.2.1 Arquitetura do motor

A arquitetura do motor é uma simplificação da arquitetura de camadas (LYRA STUDIOS, 2003), em conjunto com o modelo de publicador/assinante (FIEDLER; WALLNER; WEBER, 2002), como se pode ver na figura 5.

Figura 5 – Visão geral da arquitetura do motor

O lado servidor é composto de três camadas: sessão, localização e dados. A camada cliente, que não faz parte diretamente do projeto, é representada por implementações de jogos. O modelo publicador/assinante, por sua vez, fica distribuído entre as camadas de sessão e localização. Neste caso, a camada de localização fica responsável por gerenciar e publicar os canais, tanto de interação quanto de ambiente, enquanto a camada de sessão fica responsável por assinar e cancelar a assinatura dos canais que interessam aos clientes que estiverem conectados nela. Com isso, o modelo publicador/assinante torna-se transparente à camada cliente, afinal, todo o trabalho é efetuado nos servidores.

3.2.2 Diagramas de casos de uso

Os diagramas de casos de uso apresentados aqui representam a estrutura do protótipo a partir de dois pontos de vista: do usuário final e do engenheiro de um jogo. O segundo ator é

necessário, pois o objetivo do protótipo é ser configurável e adaptável a virtualmente qualquer tipo de jogo MMORPG. Portanto, o protótipo por si só não representa um jogo. É necessário existir um ator que defina e implemente o enredo de um jogo através das funcionalidades disponibilizadas, como os *scripts* de extensão.

Nestes casos de uso, são apresentadas apenas as atividades básicas que o cliente pode efetuar. Como o protótipo é extensível, a camada cliente também pode definir outras ações e casos de uso compatíveis com o enredo do jogo em questão, configurado através dos *scripts*.

Na figura 6, são apresentados os casos de uso do ator usuário final do jogo.

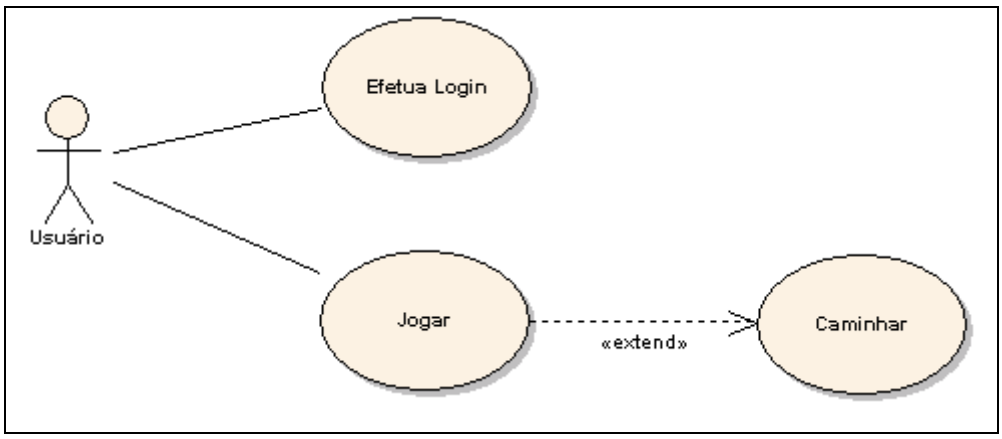


Figura 6 – Casos de uso do usuário final

O quadro 13 descreve o caso de uso efetua *login*.

EFETUA LOGIN
Descrição: Usuário do cliente do jogo solicita autenticação com o servidor para iniciar sessão de jogo. **Ator principal**: Usuário **Cenário principal**: efetua login a) através da interface disponibilizada pelo aplicativo cliente do jogo, usuário informa login e senha; b) o aplicativo cliente valida os dados e os envia para o servidor, numa solicitação de login; c) o aplicativo cliente aguarda resposta do servidor indicando se o login foi bem sucedido ou não. **Cenário de exceção**: login inválido Caso o servidor não valide as informações de login, o aplicativo cliente deve alertar o usuário que o login não foi bem sucedido. **Pré-condição**: o aplicativo cliente estar conectado com o servidor de sessão. **Pós-condição**: o aplicativo cliente estará apto a iniciar o jogo.

Quadro 13 – Caso de uso efetua login

O quadro 14 descreve o caso de uso jogar.

JOGAR
Descrição: o usuário interage, através do aplicativo cliente, com o mundo virtual, efetuando as ações disponibilizadas pelo enredo do jogo. **Ator principal**: usuário **Cenário principal**: jogar a) o usuário, através da interface disponibilizada pelo cliente, efetua as ações de jogo disponíveis pelo enredo definido; b) o aplicativo cliente envia as ações ao servidor; c) o aplicativo cliente trata os retornos das ações atualizando a interface com o usuário. **Pré-condição**: usuário conectado e login efetuado.

Quadro 14 – Caso de uso jogar

O quadro 15 descreve o caso de uso caminhar, que é uma extensão do caso de uso jogar.

CAMINHAR
Descrição: através da interface disponibilizada pelo cliente, o usuário locomove-se através do mundo virtual. **Ator principal**: usuário **Cenário principal**: caminhar a) através da interface disponibilizada pelo aplicativo cliente, e conforme a configuração de mapas do jogo, o usuário solicita que seu personagem caminhe de um ponto do mapa para outro ponto; b) cliente envia solicitação de movimentação ao servidor; c) cliente atualiza a interface do usuário conforme recebe as confirmações dos passos. **Cenário exceção**: exceção As exceções neste caso são definidas pelo enredo do jogo, através de *scripts* e impedem o usuário de caminhar. **Pré-condição**: usuário conectado e login efetuado.

Quadro 15 – Caso de uso caminhar

A figura 7 ilustra os casos de usos do engenheiro de jogo, que implementa um enredo de jogo baseado nas ferramentas disponibilizadas pelo protótipo, e é responsável pela manutenção das camadas de localização e sessão.

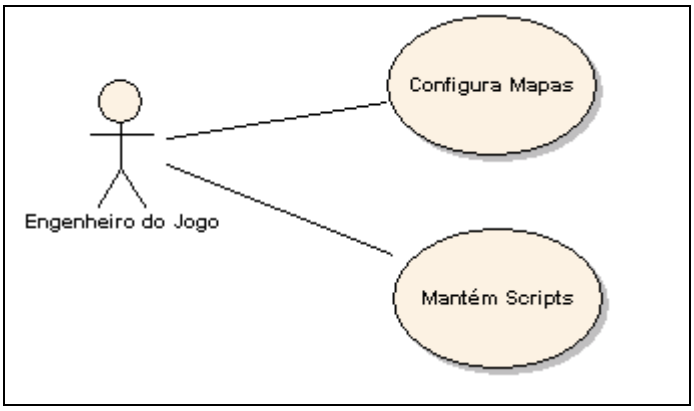


Figura 7 – Casos de uso do engenheiro de jogo

O quadro 16 descreve o caso de uso configura mapas.

CONFIGURA MAPAS
Descrição: o engenheiro configura os mapas do jogo e suas partes, bem como os servidores de localização que gerenciarão cada uma delas. **Ator principal**: engenheiro de jogo **Cenário principal**: configurar mapas a) o engenheiro configura nas tabelas do banco de dados do jogo os dados a respeito dos mapas que o jogo disponibiliza, além da maneira como são divididos; b) o engenheiro configura também as instâncias de servidores de localização que ficarão responsáveis por cada uma das partes configuradas. **Pós-condição**: mapas do mundo configurados e aptos ao uso pelos jogadores.

Quadro 16 – Caso de uso configura mapas

O quadro 17 descreve o caso de uso mantém *scripts.*

MANTÉM SCRIPTS
Descrição: O engenheiro escreve os scripts para os eventos disparados pelo protótipo, descrevendo assim o enredo do jogo **Ator principal**: Engenheiro de jogo **Cenário principal**: manter scripts a) utilizando um editor de sua preferência, o engenheiro escreve os scripts que serão disparados pelo protótipo para cada evento disparado pelo mesmos que o jogo disponibiliza, além da maneira como são divididos; b) o engenheiro armazena os arquivos de scripts conforme definição de cada tipo de servidor. **Pós-condição**: enredo do jogo configurado e jogo apto ao uso pelos jogadores.

Quadro 17 – Caso de uso mantém *scripts*

3.2.3 Diagramas de atividades

Nesta seção são demonstrados diagramas de atividades que ilustram as principais atividades executadas pelas camadas do protótipo.

Os diagramas apresentam uma visão geral dos procedimentos executados pelas camadas, bem como das operações de trocas de mensagens entre clientes, camadas de sessão e camadas de localização.

Para representar o fluxo padrão, tanto da camada de sessão quanto da camada de localização, foi criado o diagrama visível na figura 8. No fluxo individual de cada servidor, a principal parte alterada é o processamento das mensagens, já que cada um processa as mensagens conforme sua responsabilidade. Há também várias outras operações que cada um dos servidores executa. Na camada de localização, por exemplo, quem gera as mensagens que precisam ser processadas são os servidores de sessão. Já na camada de localização, são os clientes propriamente ditos e os servidores de localização.

O diagrama da figura 9 representa o fluxo geral para uma operação de autenticação de um jogador num servidor de sessão. Para este diagrama, vale lembrar que o comportamento da camada de cliente é o esperado pelo protótipo, já que ela deve ser implementada por completo por quem criar um jogo utilizando o motor do protótipo. O fluxo representado é o que deveria ser padrão, mas pode ser adaptado conforme as necessidades do cliente em questão. O diagrama também representa apenas o fluxo até o término da autenticação. Após isso, o cliente pode efetuar outras atividades, como iniciar o jogo.

Na figura 10 está representado o protocolo de solicitação de início de jogo, do ponto de vista do usuário e do servidor de sessão. Esta operação deve acontecer após a autenticação do jogador no servidor. Da mesma forma que o diagrama anterior, o fluxo do lado do cliente é o esperado pelas implementações de jogos que utilizem o servidor. Após o fluxo ser concluído com sucesso, o cliente fica apto para jogar, podendo efetuar as ações definidas pelo enredo da implementação em questão.

Por padrão, após efetuar a autenticação e solicitar o início de jogo, o cliente consegue apenas movimentar-se pelo mapa. Outras ações, como ataques, devem ser definidas através dos *scripts* de definição do enredo. O diagrama da figura 11 representa o protocolo para uma solicitação e execução de movimentação num mapa, incluindo aí a comunicação entre o cliente e o servidor de sessão, bem como a comunicação do servidor de sessão com o servidor de localização que gerencia as partes de mapas em que o usuário está se movimentando.

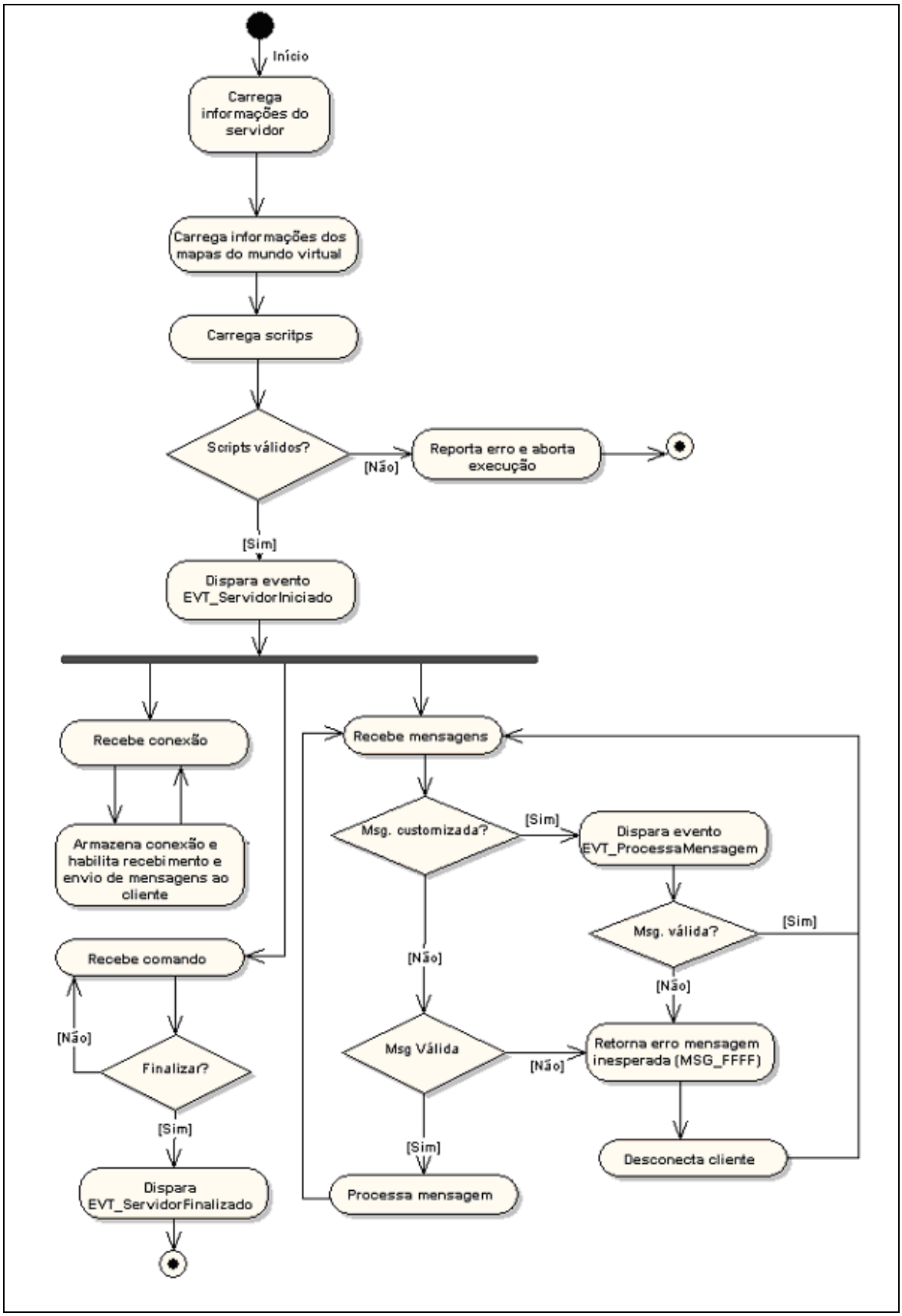


Figura 8 – Diagrama de atividades com fluxo geral das camadas do servidor

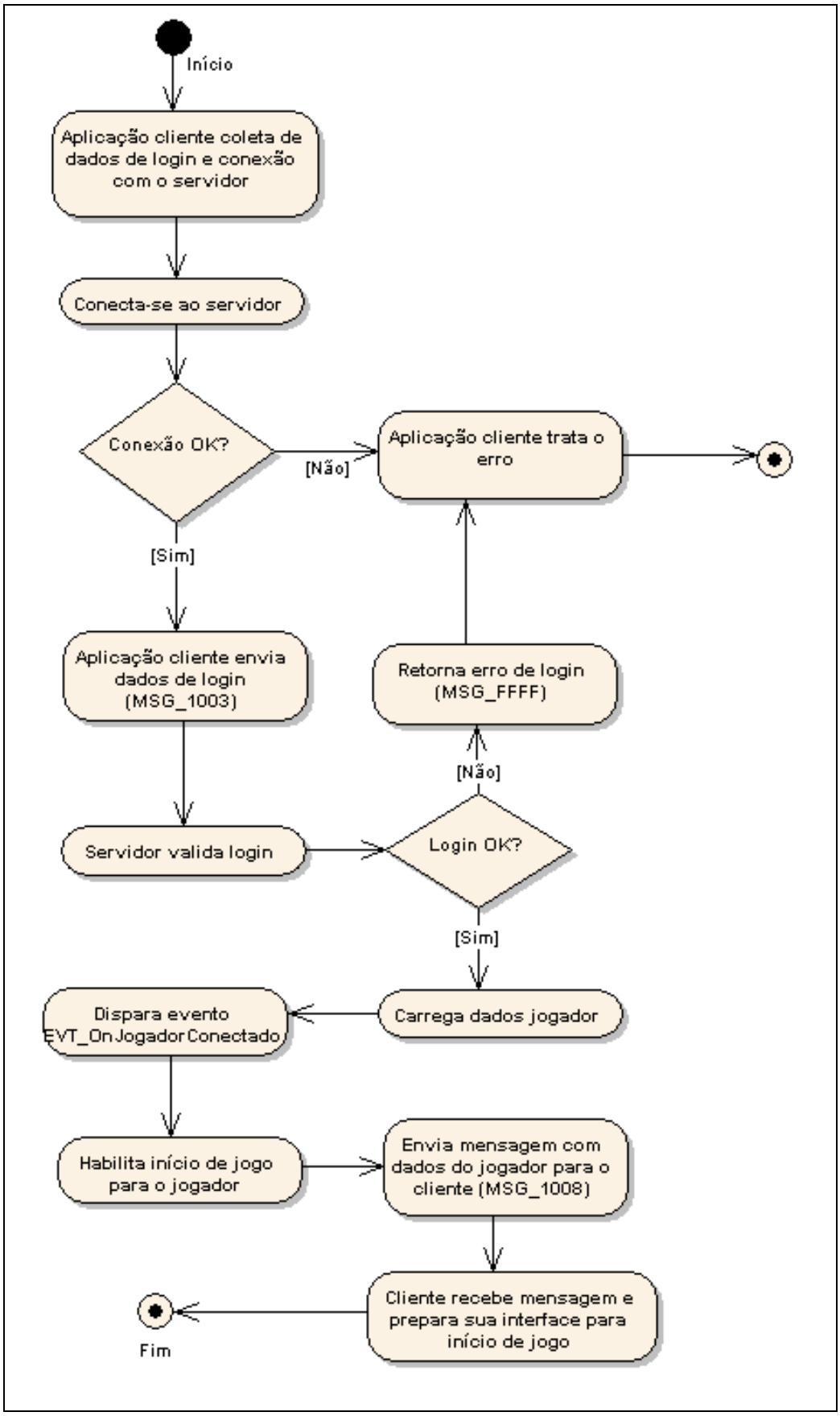


Figura 9 – Diagrama de atividades do fluxo de autenticação no servidor de sessão

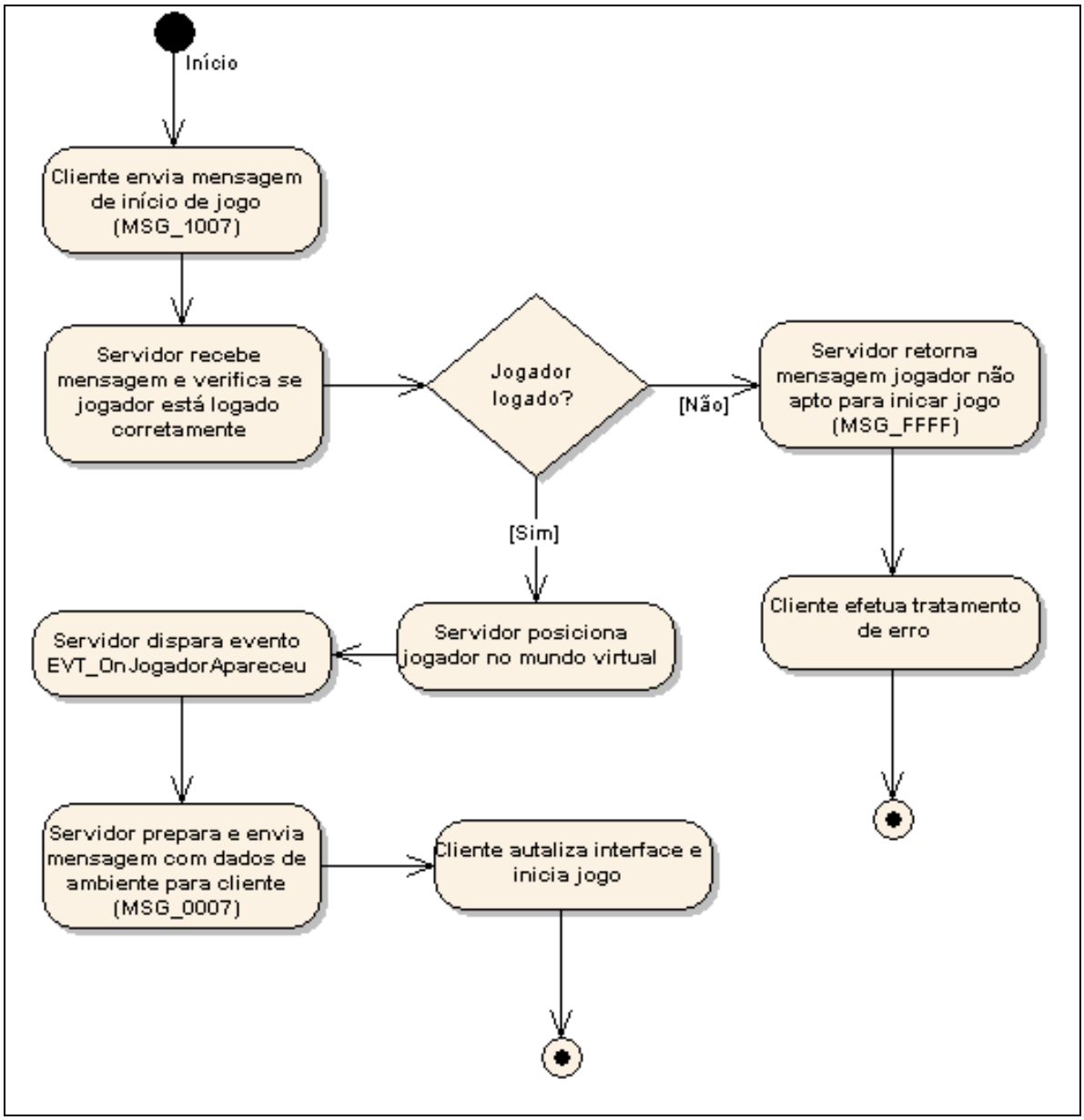


Figura 10 – Diagrama de atividades de início de jogo

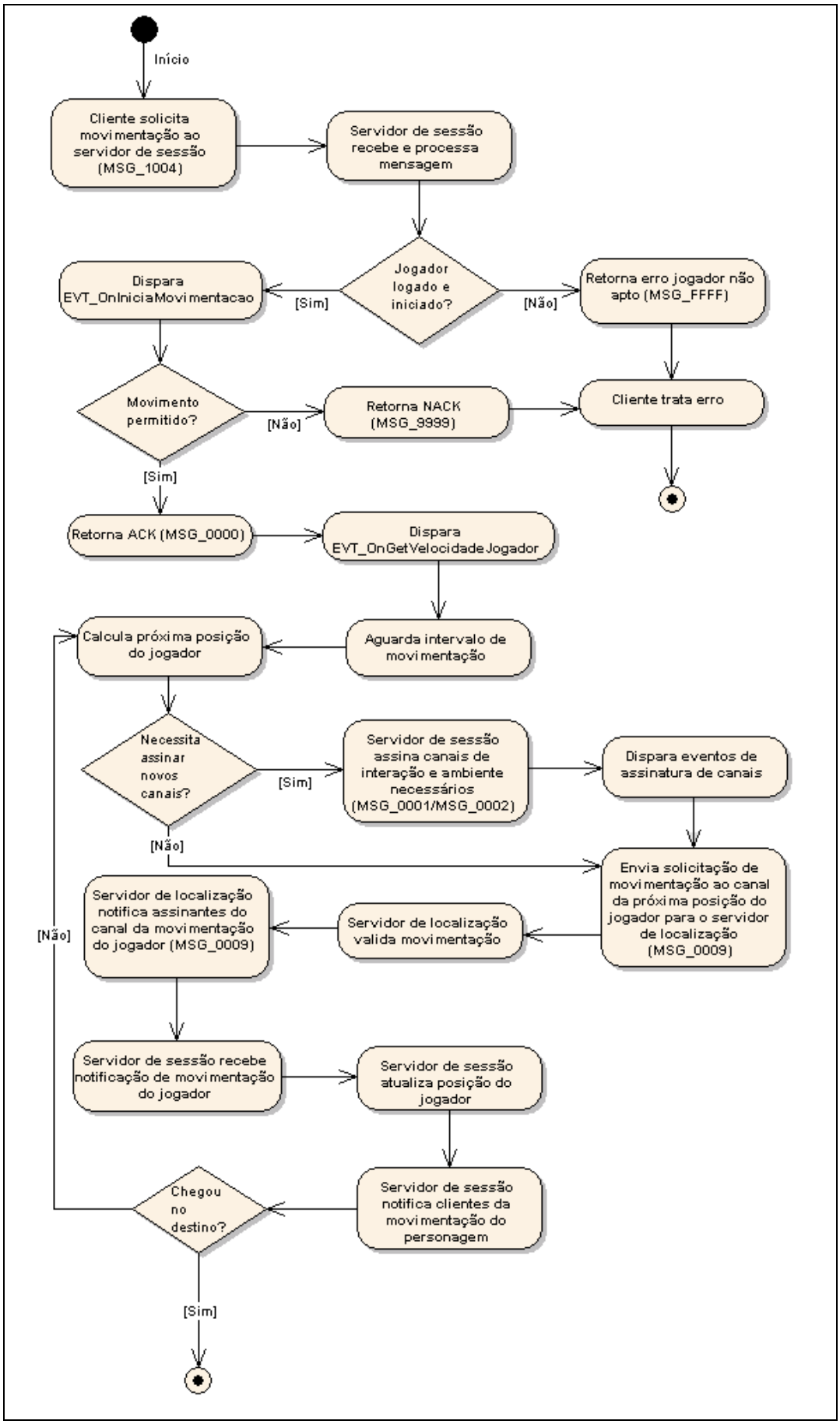


Figura 11 – Diagrama de atividades da movimentação de personagens

3.2.4 Diagrama de classes

Esta seção contém a especificação dos diagramas das principais classes das camadas do motor. Devido ao tamanho das classes, nesta seção estão os diagramas contendo apenas os principais membros de cada classe. Eles estão divididos em três partes: classes de base, classes da camada de localização e classes da camada de sessão. Após cada diagrama também é dada uma breve descrição de cada uma das classes e de seu papel no aplicativo.

A figura 12 ilustra o diagrama de classes de base. Estas classes são comuns às duas camadas do servidor, implementando as rotinas básicas suportadas por elas, como gerenciamento de conexões, gerenciamento da fila de mensagens, transformação dos *buffers* das mensagens.

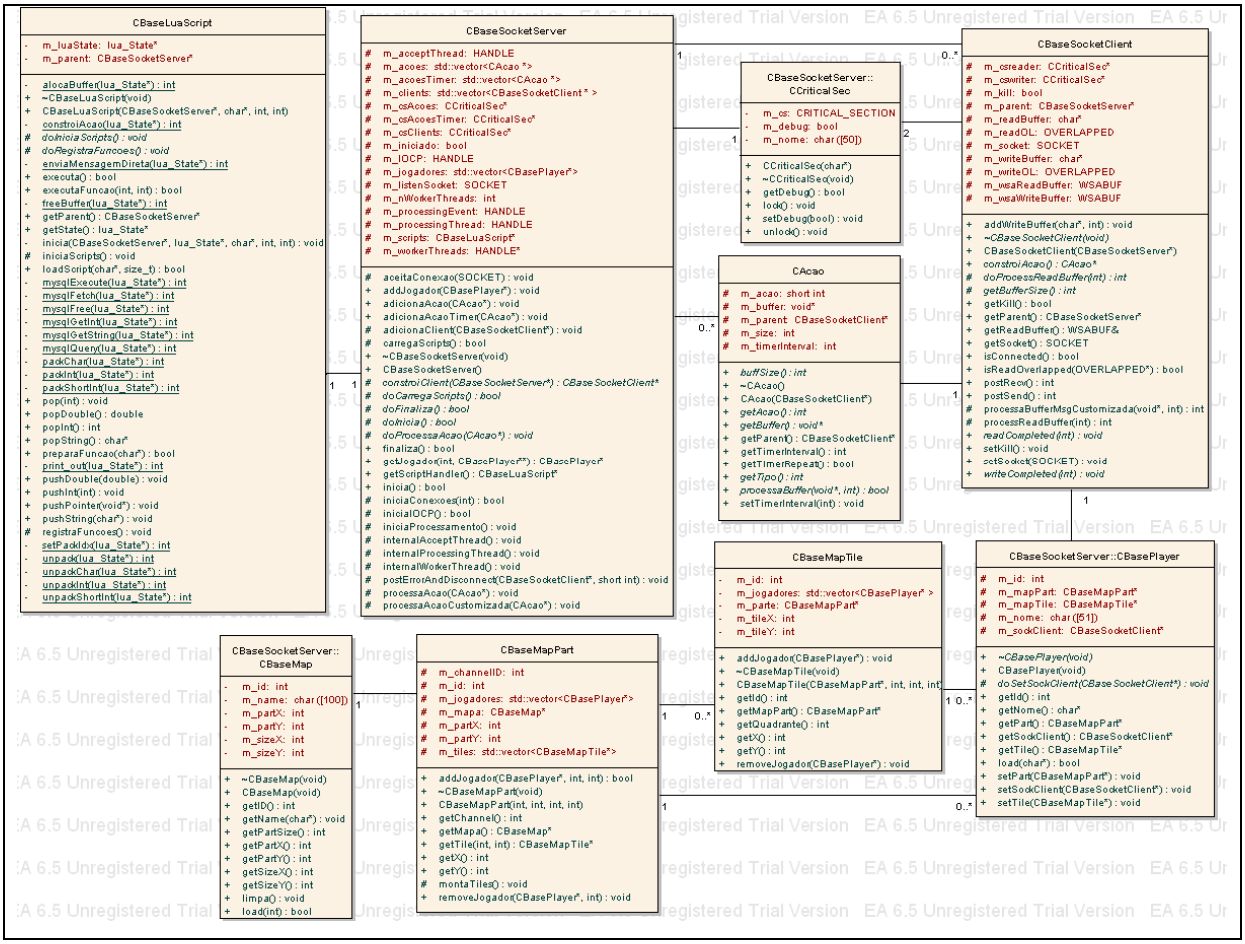


Figura 12 – Diagrama de classes base

As classes de base são as seguintes:

a) `CBaseSocketServer`: classe de base para servidores de *sockets*. Contém a implementação de suporte ao IOCP, lista de clientes e implementação dos *threads* de recebimento e gerencimaneto de conexões. Implementa também métodos

comuns ao processamento dos servidores, como transformar *buffers* lidos dos *sockets* em ações bem definidas para serem executadas pelas classes que herdam dela, através de chamadas à classe `CBaseSocketClient`. Conta também com controle de agendamento de ações, que simula um *timer*, permitindo que ações sejam executadas de tempos em tempos. Além disso também possui as rotinas de iniciação do servidor, como carga de dados, *scripts*, etc. Disponibiliza métodos virtuais para que as implementações herdeiras executem suas ações individuais durante estes passos;

b) `CBaseSocketClient`: classe de base para clientes de um objeto `CBaseSocketServer`. Representa uma conexão ao servidor, e contém a implementação das rotinas de escrita e leitura em uma conexão, bem como o controle dos *buffers* alocados para cada uma destas atividades. Contém métodos virtuais para que as classes herdeiras possam implementar suas rotinas durante a recepção e o envio de mensagens, entre outros;

c) `CAcao`: classe base para ações recebidas e/ou enviadas para o servidor. Seus objetos representam as mensagens recebidas e/ou enviadas pelo servidor, de maneira mais amigável do que os *buffers* puros utilizados pelas rotinas de *sockets*. Além disso, implementa opções que permitem ao servidor agendar as ações com o seu *timer*. Os objetos desta classe são enfileirados pela classe `CBaseSocketServer`, e são executados um a um, conforme a lógica de suas classes herdeiras;

d) `CBasePlayer`: classe base para representação de um jogador. Possui as informações padrão do jogador, como seu código, nome, posição, etc. Implementa também métodos para carga dos dados do banco de dados;

e) `CBaseMap`: classe base que representa um mapa por completo no mundo virtual do jogo. Possui as informações do mapa, como código, nome, número de partes em que se divide, tamanho de cada parte, etc. Implementa métodos para carga dos mapas do banco de dados;

f) `CBaseMapPart`: classe base que representa uma parte de um mapa do mundo virtual, e consequentemente, seus dois canais de comunicação: um de ambiente e um de interação. Possui implementação de métodos de carga das informações da parte, como sua posição no mapa completo, o servidor de localização responsável pelo gerenciamento dos canais desta parte, lista das células de posição, lista dos usuários que estão no mapa, etc;

g) `CBaseMapTile`: classe base que representa uma célula dentro de uma parte de um mapa. Estas células são as menores unidades de posicionamento dos jogadores dentro de um mapa. Possui os dados da célula, como o mapa em que está contida, sua posição, os jogadores que estão nesta posição, etc;

h) `CBaseLuaScript`: classe base para o gerenciamento de *scripts* Lua. Possui a implementação de carga, testes e execução de um *chunk* Lua, bem como passagem e recebimento de parâmetros. Além disso, implementa diversos métodos de comunicação dos *scripts* com o aplicativo servidor, dentre eles:
   - conexão com banco de dados, permitindo aos *scripts* executar sentenças *SQL* no banco de dados do jogo,
   - manipulação de *buffers*, permitindo aos *scripts* preparar *buffers* para envio de mensagens, ou processar *buffers* de mensagens recebidas por uma conexão,
   - envio de mensagens através dos servidores, para os clientes conectados,
   - recebimento de dados de ambiente dos servidores;

i) `CCriticalSec`: classe que representa uma sessão crítica, utilizada na sincronização dos *threads* do servidor. Possui facilitadores para a utilização de uma sessão crítica, bem como opções para facilitar a depuração das chamadas de entrada e saída da sessão.

As classes da camada de localização estão representadas na figura 13:

a) `CLocServer`: classe que herda de `CBaseSocketServer`, e implementa os métodos de responsabilidade do servidor de localização, como processamento das mensagens, gerenciamento dos canais das partes dos mapas, execução de *scripts* para eventos de localização, assinatura e cancelamento de assinatura de canais, etc;

b) `CLocClient`: classe que herda de `CBaseSocketClient` e implementa os métodos de conexão do servidor de localização com os servidores de sessão. Como o cliente do servidor de localização é um servidor de sessão, controla os canais assinados por cada conexão, e possui métodos para facilitar o envio de mensagens a todos os assinantes, tanto dos canais de ambiente quanto de interação;

c) `CLocPlayer`: classe que herda de `CBasePlayer`, e representa um jogador do ponto de vista do servidor de localização;

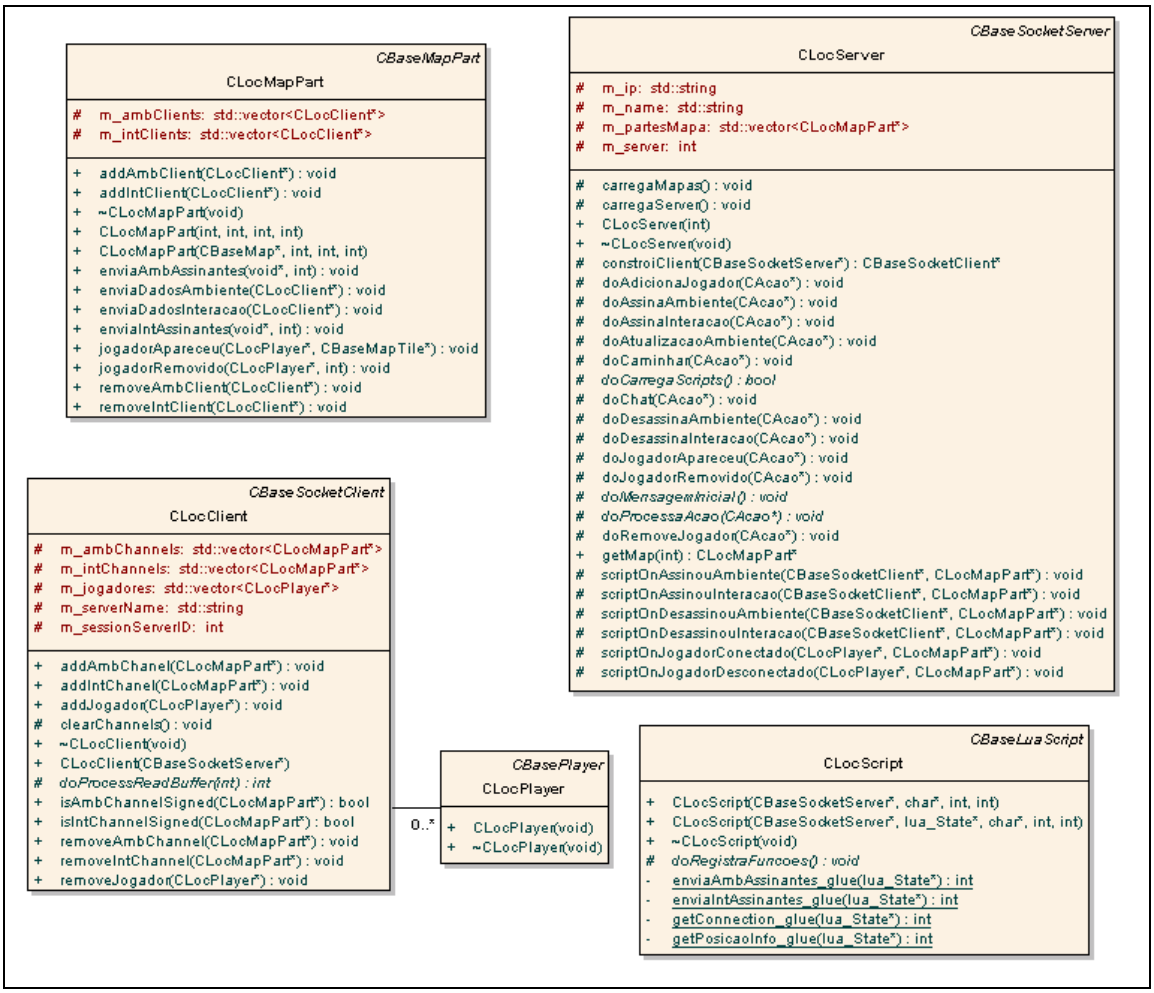


Figura 13 – Diagrama de classes de localização

d) `CLocMapPart`: classe que herda de `CBaseMapPart` e representa uma parte de um mapa do ponto de vista do servidor de localização. Representa os dois canais que são criados para cada parte de mapa, e possui a lista dos servidores de sessão assinantes de cada tipo de canal, bem como os jogadores que estão na parte em questão;

e) `CLocScript`: classe que herda de `CBaseLuaScript` e implementa os métodos de comunicação dos *scripts* especificamente com a camada de localização. Possui métodos para acesso aos jogadores que estão em uma determinada parte de mapa, bem como atalhos para envio de mensagens para os assinantes dos canais gerenciados pelo servidor de localização em questão.

As classes da camada de sessão estão representadas na figura 14:

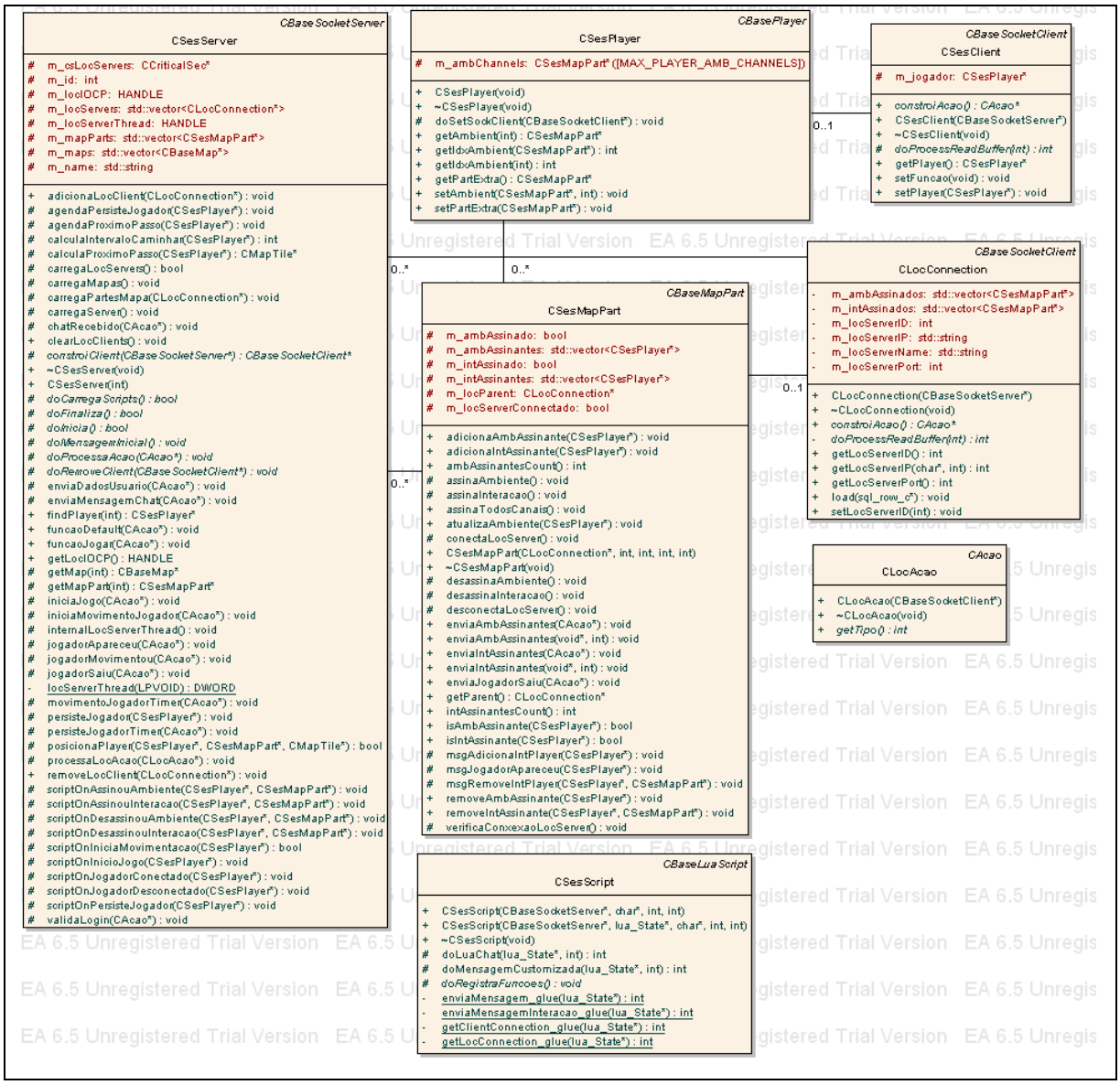


Figura 14 – Diagrama de classes de sessão

a) `CSesServer`: classe que herda de `CBaseSocketServer`, e implementa os métodos de responsabilidade do servidor de sessão, como processamento das mensagens, gerenciamento dos canais das partes dos mapas, execução de *scripts* para eventos de sessão, etc. Além disso, esta classe implementa também a conexão com os servidores de localização que gerenciam os canais dos mapas que os jogadores precisam assinar, bem como as rotinas de assinatura e cancelamento de assinatura de canais nestes servidores;

b) `CSesClient`: classe que herda de `CBaseSocketClient` e representa a conexão de um jogador com a camada de sessão. Implementa os métodos de tratamento das conexões com os jogadores, bem como o tratamento das mensagens recebidas e enviadas para os mesmos;

c) `CLocConnection`: classe que herda de `CBaseSocketClient` e representa a conexão de um servidor de localização ao servidor de sessão. Implementa os métodos de tratamento das conexões com os servidores de localização, incluindo facilitadores para assinatura de canais e publicação de mensagens nestes canais;
d) `CSesPlayer`: classe que herda de `CBasePlayer` e representa um jogador do ponto de vista da camada de sessão. Isso inclui manter um vínculo com a conexão do jogador, armazenar os canais que o jogador assina, etc;
e) `CSesMapPart`: classe que herda de `CBaseMapPart` e representa uma parte de um mapa do ponto de vista da camada de sessão. Mantém vínculo com o servidor de localização que gerencia os canais da parte em questão, a conexão com este servidor, os clientes que assinam a parte, etc;
f) `CClientAcao`: classe que herda de `CAcao` e representa uma ação efetuada ou enviada por um jogador para a camada de sessão;
g) `CLocAcao`: classe que herda de `CAcao` e representa uma ação efetuada ou enviada por um servidor de localização para a camada de sessão;
h) `CSesScript`: classe que herda de `CBaseLuaScript` e implementa os métodos de comunicação dos *scripts* especificamente com a camada de sessão, como acesso aos jogadores gerenciados pelo servidor de sessão, os canais assinados pelo servidor e seus respectivos servidores de localização, etc.

3.2.5 Protocolo de comunicação

O protocolo de comunicação é composto de mensagens de tamanho fixo, ou de tamanho variável (neste caso, com tamanho especificado em campos da própria mensagem). Desta forma, descarta-se a necessidade de marcadores de início e término das mensagens. Todas as mensagens têm um cabeçalho padrão, que contém um número de 16 bits, que indica o código da mensagem. A partir deste código o protótipo descobre o restante dos campos e, consequentemente, o tamanho da mensagem. O quadro 18 contém algumas das principais mensagens utilizadas. A descrição de todas as mensagens está no apêndice A.

Código da mensagem	Descrição	Campos
0x0001	assina canal de interação	idCanal (int - 32 bits)
0x0002	assina canal de ambiente	idCanal (int - 32 bits)
0x0003	Cancela assinatura de canal de interação	idCanal (int - 32 bits)
0x0004	Cancela assinatura de canal de ambiente	idCanal (int - 32 bits)
0x1003	Solicitação de login	loginName (char[50]) password (char[50])
0x1004	Solicitação de movimentação	mapId (int - 32 bits) tileId (int – 32)
0x0009	Confirmação de movimentação	playerId (int - 32 bits) mapId (int - 32 bits) tileId (int - 32)
0x5000	Mensagem especial de tratamento pelos *scripts* de extensão	subMsgId (int – 16 bits) msgSize (int – 16 bits) [restante dos campos definidos pelo *script* que gerencia a mensagem]

Quadro 18 – Exemplos de mensagens dos servidores

Dentre as mensagens existe uma mensagem especial, que é a mensagem utilizada pelos *scripts* de extensão para comunicação. A mensagem *0x5000* é a única do protocolo que possui tamanho variável, e, por isso, possui o campo que indica o seu tamanho (quadro 18). Ao receber uma mensagem com este cabeçalho, o servidor a encaminha diretamente à rotina de *scripts*, sem nenhum tratamento. Os *scripts*, por sua vez, ficam responsáveis por efetuar o tratamento das mensagens. Desta forma é possível estender o protocolo de comunicação, conferindo grande extensibilidade ao servidor.

3.2.6 Modelo de dados

O modelo de dados deste projeto é composto, inicialmente, por cinco tabelas, como pode ser visto na figura 15. Como o projeto é extensível, é possível que jogos que o utilizem

venham a criar novas tabelas para armazenar novos dados referentes ao enredo definido.

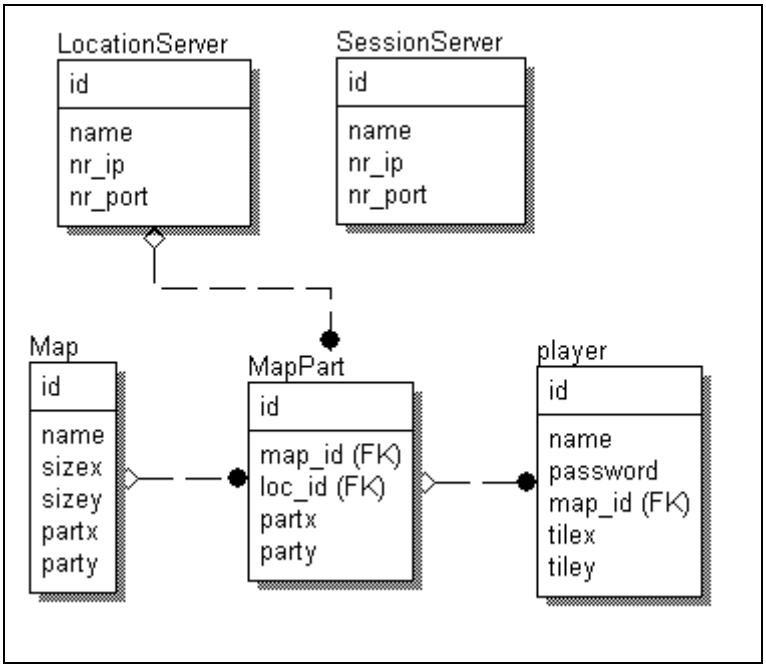


Figura 15 – Modelo de dados do protótipo

Cada banco de dados contendo estas tabelas descreve um único mundo virtual, e os propósitos das tabelas são os seguintes:

a) *LocationServer*: armazena as informações dos servidores de localização utilizados no mundo virtual, incluindo endereço e porta para conexão;
b) *SessionServer*: armazena as informações dos servidores de sessão utilizados no mundo virtual, incluindo endereço e porta para conexão;
c) *Map*: armazena as informações dos mapas que compõem o mundo virtual, incluindo o tamanho em partes (nas coordenadas X e Y, pelas colunas *sizex*, e *sizey*) e o tamanho de cada parte do mapa em células (nas coordenadas X e Y, pelas colunas *partx* e *party*);
d) *MapPart*: armazena as informações das partes dos mapas, incluindo sua posição dentro do mapa completo (coordenadas X e Y, pelas colunas *partx* e *party*), bem como o servidor de localização responsável por gerenciar os canais disponibilizados pela parte em questão (coluna *loc_id*);
e) *Player*: armazena os jogadores habilitados no mundo virtual, incluindo dados de autenticação, bem como a última posição do mesmo (colunas *map_id*, *tilex* e *tiley*), utilizada para posicionar o personagem quando um novo jogo é iniciado. Eventuais propriedades definidas pelos enredos de jogos criados para o protótipo devem, ou incluir novas colunas nesta tabela, ou definir novas tabelas para armazenar as informações.

3.2.7 Eventos disparados pelos servidores

Os eventos disparados pelos servidores têm objetivo de conferir extensibilidade ao motor, de maneira a deixar o protótipo preparado para suportar os mais variados enredos de jogos do tipo MMORPG. Estes eventos são disparados em momentos chave do jogo, de maneira a permitir que os *scripts* de extensão possam definir o comportamento do jogo. Os eventos disparados, que em grande parte são comuns às duas camadas, são os seguintes:

a) `EVT_OnIniciaServidor`: disparado no início dos servidores, antes de começar a recepção de conexões;
b) `EVT_OnJogadorConectado`: disparado após o jogador autenticar-se ao servidor, antes de iniciar o jogo propriamente dito;
c) `EVT_OnJogadorDesconectado`: disparado assim que um jogador desconecta-se do servidor;
d) `EVT_OnInicioJogo`: disparado assim que jogador inicia um jogo, após ter se autenticado ao servidor;
e) `EVT_OnPersisteJogador`: disparado assim que os dados do jogador são persistidos no banco de dados;
f) `EVT_OnIniciaMovimentacao`: disparado após receber uma solicitação de movimentação do cliente, e antes de efetuá-la. Este evento é específico à camada de sessão. Além disso, ele deve retornar um valor indicando se a movimentação é permitida (um) ou não (zero);
g) `EVT_OnGetVelocidadeJogador`: disparado após validar a movimentação do jogador, e deve retornar o intervalo de tempo entre cada passo (ou seja, troca de célula) do jogador;
h) `EVT_OnMensagem`: disparado pelos servidores ao processar uma ação da fila de ações que contenha uma mensagem específica para *scripts* (*0x5000*), no caso de uma mensagem deste tipo ter sido agendada através do sistema de *timer* do servidor;
i) `EVT_OnProcessaBufferMensagem`: disparado ao receber uma mensagem proveniente de conexão, que contenha no cabeçalho o tipo de mensagens de *scripts* (`0x5000`). Fica sob responsabilidade deste evento efetuar o tratamento do *buffer* recebido;
j) `EVT_OnAssinouInteração`: disparado quando um canal de interação é assinado.

No caso da camada de sessão, quando um jogador assina um canal destes, e no caso da camada de localização, quando um servidor de sessão assina um canal deste tipo;

k) `EVT_OnAssinouAmbiente`: idem acima, porém, quando o canal é de ambiente;
l) `EVT_OnDesassinouInteracao`: disparado quando a assinatura a um canal de interação é cancelada, tanto por um jogador (camada de sessão) quando por um servidor de sessão (camada de localização);
m) `EVT_OnDesassinouAmbiente`: idem acima, porém, quando o canal é de ambiente.

3.3 IMPLEMENTAÇÃO

Esta seção demonstra detalhes da implementação das camadas do protótipo, como as técnicas e ferramentas utilizadas, bem como a operacionalidade da implementação através de um caso de uso. Ao final, são feitos comentários sobre os resultados obtidos através do caso de uso.

3.3.1 Técnicas e ferramentas utilizadas

A implementação do protótipo foi efetuada utilizando a linguagem C++, através do ambiente Microsoft Visual Studio .Net 2003. Para a comunicação com o banco de dados foi utilizada a biblioteca MySQL C++ Warper (KENSHO, 2006). Os diagramas UML da especificação do protótipo foram realizados utilizando a ferramenta Enterprise Architect (SPARX, 2006). Já o diagrama entidade-relacionamento foi realizado com a ferramenta ERWin (ASSOCIATES, 2006).

3.3.1.1 Biblioteca MySQL C++ Warper

Trata-se de uma biblioteca bastante simples, que apenas serve como um facilitador para o acesso às bibliotecas padrão do banco de dados MySQL. Esta biblioteca disponibiliza

algumas classes, com as quais é possível efetuar as operações no banco:

a) *sql_connection_c*: representa uma conexão com um banco de dados MySQL. Em seu construtor é necessário informar os dados de conexão, como nome do servidor, usuário, banco, senha, etc;
b) *sql_query_c*: representa um *query* no banco. Através de objetos desta classe é possível executar sentenças no SQL. Caso a sentença tenha resultado, como uma sentença *SELECT*, é possível obter o objeto de resultados com ela também;
c) *sql_result_c*: representa o resultado de um *query* executado no banco de dados. Possui informações do resultado, como número de linhas e número de colunas retornadas. Através de objetos desta classe é possível carregar as linhas retornadas;
d) *sql_row_c*: representa uma linha do resultado de um *query* no banco de dados. Os objetos desta classe são, na verdade, *arrays* indexados a partir de zero contendo as colunas retornadas pelo *query*. Para facilitar a utilização, o objeto sobrescreve diversos operadores com diversos tipos de retorno, permitindo a utilização do objeto com todos os tipos primitivos de dados sem necessidade de conversão.

3.3.1.2 Implementações comuns às camadas

Como boa parte do comportamento das camadas é semelhante, foram implementadas classes de base para executar estas operações. As principais classes são a `CBaseSocketServer` e `CBaseSocketClient`, além da classe de tratamento dos *scripts* Lua, `CBaseLuaScript`.

A classe `CBaseSocketServer` implementa o gerenciamento e recebimento das mensagens do servidor, seja qual for. Já a `CBaseSocketClient` trata da comunicação propriamente dita do servidor com os clientes, sejam quais forem.

Ela é composta por 4 *threads*:

a) *thread* de recepção de conexões: fica responsável por ouvir um *socket* preparado para recepção de conexões. Ao receber uma conexão, este *thread* adiciona a conexão à lista de clientes conectados do servidor, e volta a aguardar por novas conexões. O código executado por este *thread* pode ser visto no quadro 19.

```
void CBaseSocketServer::internalAcceptThread()
{
    WSANETWORKEVENTS wsaEvents;

    //Este thread irá executar aguardando um evento de recepção de conexão até que
    //o evento m_shutdownEvent for sinalizado.
    while(WaitForSingleObject(m_shutdownEvent, 0) != WAIT_OBJECT_0)
    {
        //Aguarda o evento de recepção de conexão. Quando a aplicação receber uma conexão,
        //o evento m_acceptEvent será sinalizado.
        if (WSAWaitForMultipleEvents(1, &m_acceptEvent, FALSE, 0, FALSE) != WSA_WAIT_TIMEOUT)
        {
            //Verifica se o evento é de recepção de conexão.
            WSAEnumNetworkEvents(m_listenSocket, m_acceptEvent, &wsaEvents);
            if ((wsaEvents.lNetworkEvents & FD_ACCEPT) && (0 == wsaEvents.iErrorCode[FD_ACCEPT_BIT]))
            {
                //aceita a conexão, associando-a ao IOCP e adicionando-a
                //à lista de conexões do servidor.
                aceitaConexao(m_listenSocket);
            }
        }
    }
}
```

Quadro 19 – Código do *thread* de recepção de conexões

b) *threads* de trabalho IOCP: neste protótipo são dois, e ficam responsáveis por aguardar eventos nas conexões da fila do IOCP. As mensagens recebidas ou enviadas estão sempre vinculadas a um objeto herdeiro de *CBaseSocketClient*, que são os responsáveis por manter as informações de cada conexão. Ao receber a notificação de algum evento nas conexões, o *thread* verifica qual o tipo de evento (leitura, escrita ou erro na conexão), e invoca o método correspondente no objeto `CBaseSocketClient` da conexão em questão. Este objeto, por sua vez, efetua o devido tratamento do evento, como a criação de uma ação na fila de ações do servidor. O quadro 20 contém a implementação dos *threads* de trabalho. Os dois *threads* executam este mesmo código;

c) *thread* de processamento de ações do servidor: é responsável por verificar a fila de ações pendentes e processá-las. As ações são criadas pelos objetos `CBaseSocketClient`, ao processar mensagens recebidas dos clientes nos *threads* de trabalho IOCP. O código deste *thread* é relativamente pequeno, pois a implementação da lógica de tratamento das ações está nas classes herdeiras de `CBaseSocketServer`. Seu código pode ser visto no quadro 21.

```
void CBaseSocketServer::internalWorkerThread()
{
    int threadID = m_threadIDTemp;

    void *context = NULL;
    OVERLAPPED *overlapped = NULL;
    CBaseSocketClient *client = NULL;
    DWORD bytesTransferidos = 0;
    int bytesRecebidos = 0;
    int bytesEnviados = 0;
    DWORD bytes = 0, flags = 0;

    //thread de trabalho executa até o evento m_shutdownEvent ser sinalizado
    while (WaitForSingleObject(m_shutdownEvent, 0) != WAIT_OBJECT_0)
    {
        //aguarda evento no IOCP
        BOOL ret = GetQueuedCompletionStatus(m_IOCP,&bytesTransferidos,
            (LPDWORD)&context,&overlapped,INFINITE);
        if (context == NULL)
        {
            //se o contexto veio nulo, significa que a aplicação está sendo
            //finalizada.
            break;
        }

        //pega o CBaseSocketClient responsável pela conexão
        client = (CBaseSocketClient *)context;

        //verifica se houve erro na transmissão, ou se a conexão foi
        //finalizada pelo cliente
        if ((!ret) || ((ret) && (bytesTransferidos == 0)))
        {
            //remove o cliente
            removeClient(client);
            continue;
        }

        //verifica qual a operação que foi finalizada.
        //chama o método correspondente no objeto responsável
        //pela conexão, para que o buffer recebido/enviado
        //seja tratado
        if (client->isReadOverlapped(overlapped))
            client->readCompleted(bytesTransferidos);
        else
        {
            client->writeCompleted(bytesTransferidos);
            //se a conexão está marcada para ser desconectada,
            //fecha o socket aós finalizar o último comando
            //de escrita
            if (client->getKill())
            {
                closesocket(client->getSocket());
                client->setSocket(INVALID_SOCKET);
            }
        }
    }
}
```

Quadro 20 – Código dos *threads* de trabalho IOCP

```
void CBaseSocketServer::internalProcessingThread()
{
    //o thread de processamentod e ações executa até que o
    //evento m_shutdownEvent seja sinalizado
    while(WaitForSingleObject(m_shutdownEvent, 0) != WAIT_OBJECT_0)
    {
        //aguarda a notificação de que uma ação foi incluída na lista,
        //através da sinalização do evento m_processingEvent.
        if (WaitForSingleObject(m_processingEvent,30) == WAIT_OBJECT_0)
        {
            //desinaliza o evento, para aguardar pelo próximo.
            ResetEvent(m_processingEvent);
            CAcao* a;
            //processa todas as ações que estão na fila
            while (true)
            {
                //pega a próxima ação
                a = popAcao();
                if (a == NULL)
                    break;
                //processa ela - método implementado nos herdeiros
                processaAcao(a);
                //elimina a ação
                delete a;
                //se o timer da próxima ação agendada foi alcançado, pára
                //de processar estas ações e passa para as agendadas.
                if (getNextAcaoTimer() < GetTickCount())
                    break;
            }
        }
        //processa as ações agendadas.
        processaAcoesTimer();
    }
}
```

Quadro 21 – Código do *thread* de processamento de mensagens

Além dos *threads* citados acima, há ainda o *thread* principal do programa. Em ambas camadas, este *thread* fica responsável apenas por responder pela *interface* do programa, permitindo seu fechamento de maneira amigável.

Já a classe `CBaseSocketClient` possui a declaração dos *buffers* responsáveis pela recepção e envio de mensagens. Esta classe também é responsável por transformar os *buffers* recebidos em ações do tipo `CAcao`, que são então processadas pelo servidor. Esta transformação se dá no método `readCompleted()`, que é invocado pelos *threads* de trabalho IOCP ao finalizar uma operação de leitura.

A classe de tratamento dos *scripts*, `CBaseScripts`, por sua vez, efetua todas as rotinas de carga e comunicação da aplicação com os *scripts*. Ao iniciar o servidor, é instanciado um objeto desta classe em `CBaseSocketServer`, e o arquivo com o *script* referente à camada em questão (localização ou sessão) é carregado. Ao ser carregado, ele já é pré-processado pela

biblioteca Lua, transformando-o numa espécie de *bytecode*, que agiliza sua posterior execução. Além disso, esta classe implementa métodos de comunicação da aplicação com os *scripts*. Exemplos de declaração, e posterior registro destes métodos para utilização pelos *scripts* podem ser vistos no quadro 22. A descrição de todas as funções disponibilizadas está no apêndice C.

```
//efetua um query no banco de dados.
static int mysqlQuery_glue(lua_State* state);
//executa um comando DML
static int mysqlExecute_glue(lua_State* state);
//avança uma linha no resultado de um query
static int mysqlFetch_glue(lua_State* state);
//retorna o valor de uma coluna do tipo inteiro
static int mysqlGetInt_glue(lua_State* state);
//retorna o valor de uma o coluna do tipo string
static int mysqlGetString_glue(lua_State* state);
//libera o resultado de um query, obtido com a função mysqlQuery_glue
static int mysqlFree_glue(lua_State* state);

....
....
....

//registra as funções para serem utilizadas pelos scripts Lua
lua_register(m_luaState,"mysqlQuery",mysqlQuery_glue);
lua_register(m_luaState,"mysqlExecute",mysqlExecute_glue);
lua_register(m_luaState,"mysqlFetch",mysqlFetch_glue);
lua_register(m_luaState,"mysqlGetInt",mysqlGetInt_glue);
lua_register(m_luaState,"mysqlGetString",mysqlGetString_glue);
lua_register(m_luaState,"mysqlFree",mysqlFree_glue);
```

Quadro 22 – Declaração e registro de métodos de comunicação com *scripts* Lua

Estes métodos funcionam como *callbacks*, e são invocados automaticamente pelas bibliotecas Lua quando da execução de um *script*. Eles recebem um único parâmetro, do tipo `lua_State`. Nesta estrutura, entre outros dados, vêm os parâmetros da função invocada no *script*. O interpretador Lua não valida o número de parâmetros, ficando a cargo da função de comunicação fazer isso. Desta forma, é possível ter diversas variações nos parâmetros das funções disponibilizadas para os *scripts* Lua. O mesmo vale para os retornos, já que a sintaxe Lua suporta mais de um retorno para as funções. Os retornos, da função de comunicação podem variar, e devem ser adicionados à estrutura `lua_State` através das funções `lua_pushinteger`, `push_string`, etc. disponibilizadas pela biblioteca.

3.3.1.3 Implementação do servidor de localização

Uma vez que as classes de base executam praticamente todo o processamento das conexões, resta às classes especialistas de cada camada implementar a lógica do processamento das ações enviadas pelos seus clientes, sejam jogadores, *scripts*, ou mesmo outros servidores (no caso desta camada).

A camada de localização, através das classes `CLocServer` e `CLocClient` efetua o gerenciamento das solicitações dos servidores de sessão, bem como o gerenciamento dos canais das partes dos mapas sob sua responsabilidade. A classe `CLocClient` fica responsável pelo tratamento dos clientes do servidor de localização, efetuando a transformação das mensagens recebidas dos clientes em objetos do tipo `CAcao`, que podem ser processados pela classe `CLocServer`.

Já a classe `CLocServer` fica responsável por manter o ambiente do mundo virtual nas diversas partes de mapas alocadas para a instância do servidor, além responder às mensagens enviadas pelos servidores de sessão conectados. Em seu construtor, ela efetua a carga dos dados da instância (como nome, endereço IP, porta de conexão, etc.), além da carga dos dados das partes de mapas sob sua responsabilidade. A classe também implementa as respostas das mensagens através do método sobrescrito `doProcessaAcao()`, que pode ser visto no quadro 23. Neste quadro também é possível visualizar a quais mensagens o servidor de localização responde.

Ainda nesta classe, nos métodos de assinatura de canais (que podem ser vistos no quadro 23, nas ações `0x0001`, `0x0002`, `0X0003` e `0x0004`), são disparados os eventos de assinatura/cancelamento de assinatura de canais, conforme a mensagem recebida.

A camada de localização também contém a classe `CLocScript`, que herda de `CBaseLuaScript`, e disponibiliza mais algumas funções para os *scripts* de extensão poderem acessar dados e funções internas e específicas da camada de localização. As funções disponibilizadas podem ser vistas no quadro 24.

```
void CLocServer::doProcessaAcao(CAcao* a)
{
    //verifica qual o código da ação desejada
    switch (a->getAcao())
    {
        case 0x0001:
            //mensagem de assinatura a canal de interação
            doAssinaInteracao(a); break;
        case 0x0002:
            //mensagem de assinatura a canal de ambiente
            doAssinaAmbiente(a); break;
        case 0x0003:
            //mensagem de cancelamento de assinatura a canal de interação
            doDesassinaInteracao(a); break;
        case 0x0004:
            //mensagem de cancelamento de assinatura a canal de interação
            doDesassinaAmbiente(a); break;
        case 0x0005:
            //mensagem de adição de jogador ao servidor (carga dos dados)
            doAdicionaJogador(a); break;
        case 0x0006:
            //mensagem de remoção de jogador do servidor (liberação dos dados)
            doRemoveJogador(a); break;
        case 0x0007:
            //mensagem de que um jogador apareceu no canal de interação
            doJogadorApareceu(a); break;
        case 0x0008:
            //mensagem de que um jogador saiu do canal de interação
            doJogadorRemovido(a); break;
        case 0x0009:
            //mensagem de tratamento de movimentação do personagem
            doCaminhar(a); break;
        case 0x000A:
            //solicitação de atualização de dados de ambiente de uma parte de mapa
            doAtualizacaoAmbiente(a); break;
        case 0x000C:
            //mensagem de chat
            doChat(a);  break;
        default:
            //mensagem inesperada: desconecta o cliente
            postErrorAndDisconnect(a->getParent(),ERR_MSG_INESPERADA);
            break;
    }
}
```

Quadro 23 – Código do método *CLocServer::doProcessaAcao()*

```
//retorna posição de um jogador
static int getPosicaoInfo_glue(lua_State* state);
//retorna o objeto de conexão de um jogador
static int getConnection_glue(lua_State* state);
//envia mensagens para assinantes do canal de interação
//de uma determinada parte de mapa
static int enviaIntAssinantes_glue(lua_State* state);
//envia mensagens para assinantes do canal de ambiente
//de uma determinada parte de mapa
static int enviaAmbAssinantes_glue(lua_State* state);
```

Quadro 24 – Funções de comunicação com *scripts* Lua

Na aplicação final do servidor de localização, a *interface* é do tipo console (sem janelas, apenas texto). Para indicar qual o servidor de localização que a instância do

executável deve gerenciar, utiliza-se um argumento na linha de comando da chamada ao executável com o código do servidor na tabela de servidores de localização do banco de dados. Para finalizar a aplicação, basta pressionar qualquer tecla em sua janela.

3.3.1.4 Implementação do servidor de Sessão

Nesta camada as principais classes são `CSesServer`, `CSesClient` e `CLocConnection`. A primeira classe é responsável por manter o servidor de sessão, gerenciando tanto os jogadores conectados a ele, quanto as conexões efetuadas com os servidores de localização responsáveis pelas partes de mapas assinadas pelos jogadores. Como existem dois tipos de conexões que este servidor faz (com clientes e outros servidores de localização), existem também duas classes para gerenciar estas conexões. As conexões com os jogadores são representadas pela classe `CSesClient`, e as conexões com os servidores de localização são representadas pela classe `CLocConnection`.

Como a classe ancestral de `CSesServer` consegue gerenciar apenas um tipo de conexões – no caso, as dos jogadores – , esta classe precisa implementar o gerenciamento das conexões com os servidores de localização. Entretanto, este gerenciamento é simples e limita-se à criação de mais um IOCP para as conexões com os servidores de localização, e mais um *thread* de trabalho responsável por aguardar eventos neste IOCP. As mensagens recebidas, são processadas da mesma forma que a classe `CBaseSocketServer` faz, e quando há ações resultantes, elas são adicionadas à fila de ações, que é comum aos dois tipos de conexões.

A separação destas ações se dá no método sobrescrito `doProcessaAcao()` de `CSesServer`, como pode ser visto no quadro 25. Neste quadro pode-se perceber que, caso a mensagem seja proveniente de um servidor de localização, há um método específico para o tratamento. Já no caso de ações vindas de jogadores, há uma propriedade no objeto `CSesClient` (responsável pela conexão do jogador) com um ponteiro para o método que deve ser utilizado. Isso se dá para facilitar a implementação, pois há dois momentos distintos no processamento das mensagens de um jogador: a autenticação e as ações de jogo. Da mesma forma, há dois métodos, responsáveis por cada um destes momentos.

```
void CSesServer::doProcessaAcao(CAcao* a)
{
    //verifica qual o tipo de ação a ser processada
    if (a->getTipo() == 2) // Ação de jogadores
    {
        //pega o objeto de gerenciamento da conexão que
        //originou a ação
        CSesClient* client = (CSesClient*)a->getParent();

        //executa o método de tratamento da ação que está
        //configurado no objeto
        (*this.*(client->getFuncao()))(a);
    }
    else
    if (a->getTipo() == 1) // Ação de servidores de localização
    {
        //processa a ação do servidor de localização
        processaLocAcao((CLocAcao*) a);
    }
}
```

Quadro 25 – Código de *CSesServer::doProcessaAcao()*

Após um cliente conectar-se, o ponteiro do método responsável pelo processamento das ações é iniciado com a função de autenticação. Este método é visível no quadro 26, onde também é possível ver quais são as mensagens suportadas por ele. Caso o jogador envie uma mensagem de jogo, antes de efetuar a autenticação, por exemplo, será desconectado com erro de mensagem inesperada.

```
void CSesServer::funcaoDefault(CAcao* a)
{
    switch (a->getAcao())
    {
        case 0x1003:
            //solicitação de login
            validaLogin(a); break;
        case 0x1006:
            //solicitação de envio de dados do usuário
            enviaDadosUsuario(a); break;
        case 0x1007:
            //solicitação de início de jogo
            iniciaJogo(a);  break;
        default:
            postErrorAndDisconnect(a->getParent(),ERR_MSG_INESPERADA);
            break;
    }
}
```

Quadro 26 – Método de tratamento de mensagens de autenticação

Após a autenticação ter sido efetuada, e o cliente enviar a mensagem de início de jogo (*0x1007*), o método de tratamento é alterado para o responsável pelas mensagens de jogo, que

pode ser visto no quadro 27. Da mesma forma, é possível ver quais as mensagens suportadas por este método.

Já o método de tratamento das mensagens recebidas dos servidores de localização pode ser visto no quadro 28, juntamente com as mensagens suportadas por ele.

```
void CSesServer::funcaoJogar(CAcao* a)
{
    switch (a->getAcao())
    {
        case 0x1004:
            //solicitação de início de movimentação de personagem
            iniciaMovimentoJogador(a); break;
        case 0x0009:
            //efevtivação de movimentação de personagem
            movimentoJogadorTimer(a); break;
        case 0x1005:
            //troca de mensagens por chat
            enviaMensagemChat(a); break;
        case 0x1009:
            //persistência do jogador efetuada de tempos em tempos
            persisteJogadorTimer(a); break;
        default:
            postErrorAndDisconnect(a->getParent(),ERR_MSG_INESPERADA);
            break;
    }
}
```

Quadro 27 – Método de tratamento de mensagens de jogo

```
void CSesServer::processaLocAcao(CLocAcao* a)
{
    switch (a->getAcao())
    {
        case 0x0007:
            //notificação de jogador aparecido num canal
            jogadorApareceu(a); break;
        case 0x0008:
            //notificação de jogador eliminado de um canal
            jogadorSaiu(a); break;
        case 0x0009:
            //notificação de movimentação de um jogadnor nun canal
            jogadorMovimentou(a); break;
        case 0x000C:
            //chat recebido
            chatRecebido(a); break;
        default:
            postErrorAndDisconnect(a->getParent(),ERR_MSG_INESPERADA);
            break;
    }
}
```

Quadro 28 – Método de tratamento de mensagens dos servidores de localização

A camada de sessão também possui uma implementação de classe de *scripts*, chamada `CSesScript`. Da mesma forma que na camada de localização, esta classe disponibiliza métodos específicos da camada de sessão para serem utilizados pelos *scripts* Lua. Além disso, ela também dispara os eventos especificados para a camada, conforme as mensagens são tratadas. Um exemplo de evento pode ser visto no quadro 29. O método representado neste quadro é o responsável por receber uma solicitação de movimentação do cliente e encaminhar as ações de movimentação para o servidor de localização responsável pelo mapa em que o jogador se encontra. Antes de efetuar a movimentação, é disparado o evento `EVT_OnIniciaMovimentacao`, através do método `scriptOnIniciaMovimentacao()`. Conforme o retorno do *script*, o servidor permite ou não a caminhada.

Da mesma forma que no servidor de localização, o executável é do tipo console. Para indicar a uma instância da aplicação qual servidor ela deve gerenciar, deve-se passar como argumento na linha de comando do executável, o código do servidor na tabela de servidores do banco de dados. Para finalizar a instância, basta pressionar qualquer tecla na janela da aplicação.

```
void CSesServer::iniciaMovimentoJogador(CAcao* a)
{
    //carrega o buffer para uma estrutura MSG_1004
    //que possui os campos definidos
    MSG_1004* msg = (MSG_1004*)a->getBuffer();

    //carrega o mapa de destino da solicitação de movimentação
    CSesMapPart* mapDest = getMapPart(msg->mapId);

    //se o mapa for inválido, retorna erro.
    if (!mapDest)...
    //carrega a célula de detitno
    CMapTile* tileDest = mapDest->getTile(msg->tileId);

    //se a célula for inválida, retorna ero.
    if (!tileDest)...

    //carrega o jogador que quer movimentar
    CSesPlayer* jog = ((CSesClient*)a->getParent())->getPlayer();

    //se o jogador for inválido, retorna erro.
    if (!jog)...

    //se a posição final for a posição atual, não há o que fazer
    if (jog->getPart()->getChannel() == msg->mapId &&
        jog->getTile()->getId() == msg->tileId)
        return;
    bool isWalking = jog->isWalking();
    bool scriptOk = true;
    //se o jogador não está caminhando, quer dizer que está iniciando
    //a operação agora. Verifica com o script se o jogador pode se movimentar.
    if (!isWalking)
        scriptOk = scriptOnIniciaMovimentacao(jog);
    //se o jogador pode se movimentar, seta o destino dele
    //e retorna ACK ao cliente. Se não, envia NACK
    if (scriptOk)
    {
        jog->setWalkDest(tileDest);
        a->getParent()->enviaAck(msg->idMsg);
    }
    else
    {
        a->getParent()->enviaNack(msg->idMsg);
        return
    }
    //se não estiver caminhando, agenda o próximo passo.
    if (!isWalking)
        agendaProximoPasso(jog);
}
```

Quadro 29 – Tratamento de mensagem de solicitação de movimentação

3.3.2 Operacionalidade da implementação

Esta seção descreve a funcionalidade do protótipo através de um estudo de caso onde é

implementado, através dos *scripts* de extensão, um enredo de jogo em que jogadores podem atacar uns aos outros. Os jogadores também possuem um atributo chamado *Health Points* (HP) que indica quantos ataques podem receber antes de sua energia acabar e não poderem mais jogar. Além disso, como este projeto contempla apenas o lado do servidor, para poder efetuar os testes foi necessário implementar um cliente com *interface* simples para facilitar a visualização do mundo virtual por parte do jogador.

3.3.2.1 Configurações do mundo virtual

O mundo virtual deste exemplo foi configurado como apenas um grande mapa, dividido em oito partes. Existem duas instâncias de servidor de localização, chamados “LOC1” e “LOC2” e cada uma fica responsável por gerenciar quatro partes de mapas. Cada parte do mapa, por sua vez, é dividida em 100 células, que representam as posições que um jogador pode assumir dentro de uma parte de mapa. A figura 16 representa a visão geral deste mundo virtual, inclusive a divisão das partes de mapas para cada um dos servidores de localização.

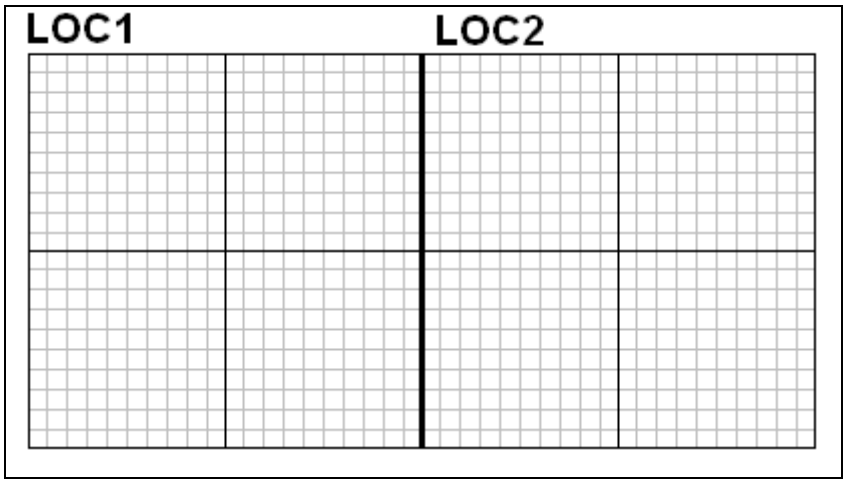


Figura 16 – Representação do mundo virtual do teste

Existem também duas instâncias de servidores de sessão, chamadas “SES1” e “SES2”. Como todas as instâncias são executadas numa única máquina, estão configuradas com o mesmo endereço IP (127.0.0.1), porém, com portas diferentes. Os servidores de localização “LOC1” e “LOC2” ficam com as portas 2020 e 2121 respectivamente, enquanto os servidores de sessão “SES1” e “SES2” ficam com as portas 3030 e 3131. Como não foi definida uma *interface* para estas configurações, elas foram feitas diretamente no banco de dados da aplicação.

O enredo deste teste define o novo atributo HP aos jogadores. Entretanto, não há, na especificação do protótipo, onde armazenar este dado. Portanto, para armazenar este atributo, foi criada uma nova tabela no banco de dados, chamada *player_userdata* (figura 17). Os *scripts* de extensão, por sua vez, ficam responsáveis por gravar e carregar os dados dos jogadores de lá, através dos eventos disparados pelos servidores, demonstrando a extensibilidade deste protótipo.

Da mesma forma, as ações de ataque de um jogador a outro foram definidas através de *scripts*. Neste caso, o cliente que foi criado está preparado para enviar mensagens especiais ao servidor de sessão, que serão tratadas pelos *scripts* dos eventos de recebimento de mensagens. Estes *scripts* estão disponíveis por completo no anexo A.

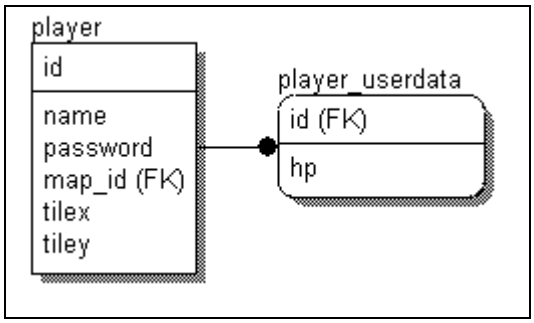


Figura 17 – Tabela para armazenamento do atributo HP dos jogadores

A figura 18 demonstra a *interface* cliente criada para representar o jogo. Na *interface*, a grade define os pontos em que é possível movimentar-se no mundo virtual. Os jogadores são representados pintando a célula em que estão posicionados. A célula em azul é o jogador propriamente dito, as em vermelho são outros jogadores, e as em cinza são jogadores derrotados numa luta. As ações do jogador se dão através de cliques nas células do mundo virtual. Ao clicar numa célula vazia, o personagem faz o processo de movimentação:

a) envia uma mensagem ao servidor de sessão solicitando movimento até o local desejado;
b) aguarda retorno do servidor;
c) conforme as mensagens de atualização de posição do jogador, a *interface* é atualizada.

Já a ação de atacar outro jogador acontece ao clicar na célula em que o adversário se encontra. Devido à maneira que os *scripts* de enredo deste exemplo foram implementados, é necessário que o jogador e seu adversário estejam em células adjacentes. Caso contrário, o ataque falha. Ao atacar, a aplicação cliente faz o seguinte processo:

a) envia uma mensagem ao servidor de sessão solicitando ataque ao adversário;
b) aguarda retorno do servidor;

c) ao receber os retornos, exibe as mensagens de ataque e atualiza a quantidade de HP restante no adversário.

A aplicação cliente também conta com uma área onde é possível visualizar as mensagens que estão sendo trocadas com o servidor. No exemplo, pode-se ver uma mensagem de atualização de posição (MSG_0009), seguida de uma mensagem de *chat*. As mensagens são exibidas com formatação hexadecimal. Estas mensagens também são salvas num arquivo de *log*, contendo as mensagens recebidas e enviadas, bem como a hora em que aconteceram. Desta forma é possível analisar a funcionalidade do servidor, verificando se ele respeita os intervalos de movimentação e ataque definidos.

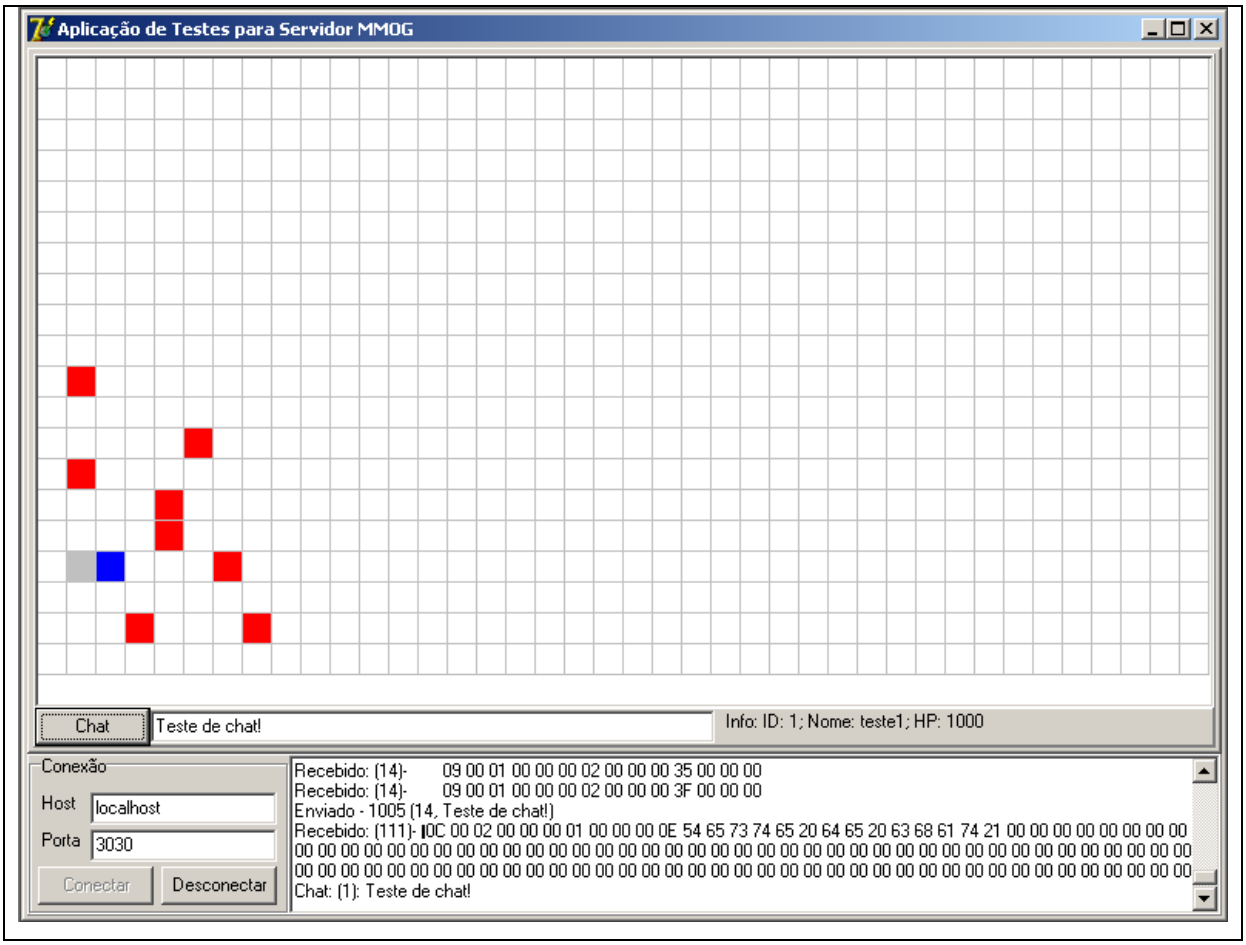


Figura 18 – Aplicação cliente de testes

Já os servidores, que são do tipo console, exibem apenas as mensagens de inicialização. Apenas no caso de erros em *scripts* ou outras operações, as *interfaces* recebem alguma atualização, normalmente com a mensagem de erro. As figuras 19 e 20 ilustram uma instância de servidor de localização e uma de sessão, respectivamente.

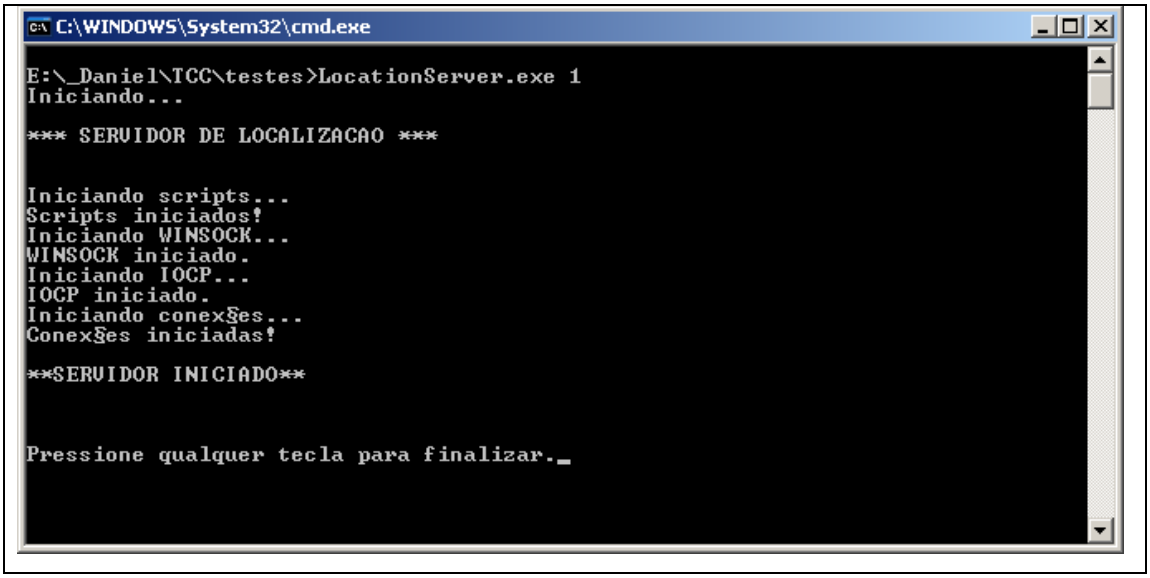


Figura 19 – Interface de uma instância do servidor de localização

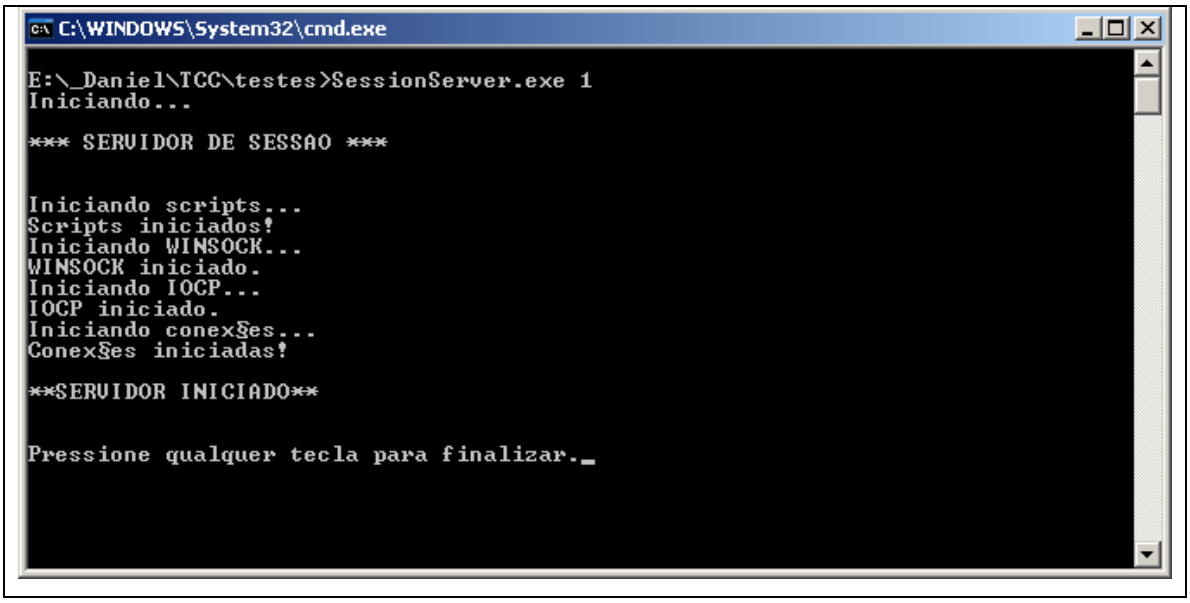


Figura 20 – Interface de uma instância do servidor de sessão

3.4 RESULTADOS E DISCUSSÃO

O protótipo foi validado através da execução de diversos testes na *interface* definida para o caso de uso descrito na seção anterior. Os testes consistiram na execução de diversas operações sobre o ambiente montado, e observação dos resultados através da *interface* disponibilizada, dos *logs* gerados pelos clientes, e das eventuais mensagens geradas pela *interface* dos servidores.

Inicialmente, foram conectados diversos clientes a ambas as instâncias do servidor de

sessão que foram executadas. Todos os outros jogadores que estavam na mesma área, independente do servidor de sessão em que estavam conectados, eram atualizados corretamente conforme os novos clientes eram conectados. Devido a restrições de hardware (apenas duas máquinas disponíveis para os testes), foi possível conectar apenas 45 clientes.

Após todos os clientes serem conectados, iniciou-se os testes de movimentação dos jogadores. Da mesma forma que na conexão, conforme os jogadores movimentavam-se, todos os outros clientes eram atualizados corretamente com as novas posições do jogador em movimento, independente do servidor de sessão em que se encontravam. Da mesma forma, conforme o jogador movimentava-se, era possível verificar as mensagens de jogador incluído e jogador removido ao entrar em áreas que exigissem assinatura de canais em outra instância de servidor de localização. Isso foi verificado observando a movimentação de jogador através de um cliente diferente do que estava movimentando-se.

O próximo teste consistiu no ataque a outro jogador. Inicialmente tentou-se atacar um jogador que estava a mais de uma célula de distância. O ataque não foi bem sucedido, e foi recebida a mensagem correspondente do servidor (mensagem esta, gerada a partir dos *scripts* de extensão). Após isso, posicionou-se o jogador corretamente e o teste foi repetido. Desta vez, o ataque foi bem sucedido e as mensagens de ataque foram recebidas até que o HP da vítima terminasse. Tentou-se, então, movimentar o jogador que havia sido derrotado a partir do cliente com o qual estava conectado, mas não foi possível, conforme a definição do enredo de testes.

Depois deste teste, para garantir que os canais e servidores estavam sendo acessados corretamente, foi feito um teste de finalizar um dos servidores de localização. O servidor finalizado foi “LOC2”, e, inicialmente, nenhum efeito foi visível, tanto nos servidores de sessão quanto nos clientes. No entanto, ao tentar movimentar um jogador que estava posicionado numa das partes que eram gerenciadas pelo servidor finalizado, o servidor de sessão começou a apresentar mensagens de erro de conexão. Entretanto, ao tentar movimentar um cliente que estava numa área do servidor de localização “LOC1” e que não assinava nenhum canal do servidor “LOC2”, a movimentação aconteceu sem problemas.

Após isso, iniciou-se a verificação dos *logs* de mensagens gerados. Neles foi possível verificar que todas as mensagens que eram esperadas foram recebidas (assim como foi possível verificar isso pelo resultado apresentado na *interface* do cliente). Além disso, foi possível verificar se o intervalo entre as mensagens de movimentação e de ataque estavam de acordo com o que havia sido configurado nos *scripts* de extensão. O intervalo entre cada passo de movimentação estava configurado para 200 milissegundos. Já o intervalo entre os

ataques, estava configurado para 150 milisegundos. Verificando os resultados, identificou-se que, apesar de pequenas variações, o intervalo entre as mensagens recebidas do servidor seguiu o padrão definido. Um exemplo dos resultados obtidos pode ser observado nos quadros 30 e 31. O primeiro demonstra as mensagens recebidas de movimentação de personagens, e o segundo com as mensagens recebidas de ataques a personagens. Cada linha do *log* representa uma operação. Os mnemônicos TX e RX representam o tipo de operação, seguidos do *TickCount* do momento em que a operação aconteceu (contado em milissegundos), do tamanho da mensagem e finalmente da mensagem propriamente dita, com os *bytes* representados por seus valores ASCII.

```
TX (19487890) (10): [04 10 03 00 00 00 3F 00 00 00 ]
RX (19487890) (4): [00 00 04 10 ]
RX (19488109) (14): [09 00 01 00 00 00 02 00 00 00 0B 00 00 00 ]
RX (19488359) (14): [09 00 01 00 00 00 02 00 00 00 15 00 00 00 ]
RX (19488609) (14): [09 00 01 00 00 00 02 00 00 00 1F 00 00 00 ]
RX (19488843) (14): [09 00 01 00 00 00 02 00 00 00 29 00 00 00 ]
RX (19489093) (14): [09 00 01 00 00 00 02 00 00 00 33 00 00 00 ]
RX (19489343) (14): [09 00 01 00 00 00 02 00 00 00 3D 00 00 00 ]
RX (19489593) (14): [09 00 01 00 00 00 02 00 00 00 47 00 00 00 ]
RX (19489828) (14): [09 00 01 00 00 00 02 00 00 00 51 00 00 00 ]
RX (19490078) (14): [09 00 01 00 00 00 02 00 00 00 5B 00 00 00 ]
RX (19490312) (14): [07 00 01 00 00 00 03 00 00 00 01 00 00 00 ]
RX (19490562) (14): [09 00 01 00 00 00 03 00 00 00 0B 00 00 00 ]
RX (19490812) (14): [09 00 01 00 00 00 03 00 00 00 16 00 00 00 ]
RX (19491062) (14): [09 00 01 00 00 00 03 00 00 00 20 00 00 00 ]
RX (19491296) (14): [09 00 01 00 00 00 03 00 00 00 2A 00 00 00 ]
RX (19491531) (14): [09 00 01 00 00 00 03 00 00 00 35 00 00 00 ]
RX (19491765) (14): [09 00 01 00 00 00 03 00 00 00 3F 00 00 00 ]
```

Quadro 30 – Resultado do *log* de uma operação de movimentação no cliente

```
TX (20244734) (14): [00 50 02 50 0E 00 01 00 00 00 02 00 00 00 ]
RX (20244890) (18): [00 50 03 50 12 00 01 00 00 00 02 00 00 00 64 00 00 00 ]
RX (20245046) (18): [00 50 03 50 12 00 01 00 00 00 02 00 00 00 64 00 00 00 ]
RX (20245203) (18): [00 50 03 50 12 00 01 00 00 00 02 00 00 00 64 00 00 00 ]
RX (20245359) (18): [00 50 03 50 12 00 01 00 00 00 02 00 00 00 64 00 00 00 ]
RX (20245515) (18): [00 50 03 50 12 00 01 00 00 00 02 00 00 00 64 00 00 00 ]
```

Quadro 31 – Resultado do *log* de uma operação de ataque a outro jogador

4 CONCLUSÕES

O protótipo implementado representa um exemplo de servidor de jogos *online* em massa. Ele possui boa parte das características esperadas de uma aplicação deste tipo, como a escalabilidade proporcionada pelo modelo de publicador-assinante juntamente com a utilização dos IOCP, a extensibilidade disponibilizada através da integração com o interpretador de *scripts* Lua, e a disponibilidade e tolerância a falhas conseguida com a separação das camadas no servidor.

Entretanto, uma aplicação como um servidor de jogos *online* em massa não é uma tarefa trivial, nem mesmo uma tarefa que possa ser executada por uma única pessoa. Servidores comerciais são resultados de times de profissionais de diversas especializações trabalhando por vários meses, e até anos, unicamente num produto deste tipo.

Ainda assim, as técnicas e ferramentas utilizadas demonstraram um resultado satisfatório aos requisitos definidos para este projeto. Além disso, o grau de extensibilidade conseguido permite que muitas outras funcionalidades sejam implementadas, sendo possível conceber enredos relativamente complexos e que apresentem muitas outras características de MMORPGs além das alcançadas no caso de uso implementado.

Dos objetivos e requisitos declarados para este projeto, exceto pela implementação de uma camada cliente no formato de API, todos foram alcançados. A camada cliente acabou sendo desconsiderada pois, além de se distanciar do escopo do trabalho, provou-se uma ferramenta desnecessária, já que o protocolo especificado ficou simples o suficiente. Além disso, uma camada cliente destas, no caso da implementação de um jogo real, tornaria-se um problema por permitir que clientes não autorizados fossem implementados com muita facilidade. Clientes não autorizados, para jogos *online* podem ser um problema por darem vantagens desleais aos jogadores que os utilizam. Dessa forma, embora seja tecnicamente impossível impedir que seja criado um cliente que simule o cliente oficial, não é interessante facilitar isso.

Embora seja difícil encontrar parâmetros para comparação, já que os motores de jogos existentes são proprietários e têm seus dados mantidos em relativo sigilo, as vantagens apresentadas pelo protótipo estão na sua própria arquitetura. A utilização de IOCP se demonstrou satisfatória, tanto do ponto de vista do desempenho, quanto do gerenciamento propriamente dito das conexões. A linguagem de *scripts* Lua se mostrou poderosa o suficiente para aplicações deste tipo, e a arquitetura de publicador-assinante, além de restringir

satisfatoriamente o tráfego de dados, garantiu a escalabilidade e tolerância a erros esperada. Além disso, a simples criação de uma aplicação como esta já mostra-se como um desafio vencido, já que há muito pouco conhecimento estabelecido sobre os assuntos relacionados.

No entanto, o protótipo desenvolvido possui algumas limitações, como as listadas a seguir:

a) não permite inimigos que não sejam outros jogadores, e nem dá possibilidades de implementação disto através dos *scripts* de extensão;
b) não permite a definição de mapas em formatos que não retangulares, e a definição de áreas não transitáveis no mapa precisa ser definida com *scripts*;
c) não possui *interface* para configurações do servidor, como mapas, servidores de sessão e localização, etc., sendo que estas configurações precisam ser feitas diretamente no banco de dados;
d) o sistema de autenticação não é seguro, pois a senha trafega aberta até o servidor;
e) embora a tolerância a erros seja garantida pela arquitetura do servidor, a implementação das camadas não trata este tipo de erros adequadamente. Com isso, caso algum erro aconteça, normalmente o ambiente só se reestabelece de maneira adequada após o reinício de todos os servidores;
f) não há ferramentas nem a figura de um administrador do servidor, portanto, a única maneira de desconectar um cliente, por exemplo, é finalizando os servidores;
g) a mensagem de *chat* possui tamanho fixo, fazendo transitar dados desnecessários em mensagens curtas;
h) o gerenciamento de memória e de sincronização entre os *threads* não é eficiente. Para memória, aloca-se e libera-se conforme a necessidade, sendo que o recomendável seria utilizar algum tipo de *pool*. Já para a sincronização, utiliza-se sessões críticas, que também poderiam ser substituídas por alternativas mais rápidas, embora mais complexas.

4.1 EXTENSÕES

Por tratar-se de um protótipo de uma aplicação do tamanho de um servidor de jogos *online*, existem diversas extensões que podem ser sugeridas. No entanto, entre elas destacam-se:

a) criação de inimigos que não necessariamente sejam outros jogadores, como monstros;
b) criação de ferramentas de gerenciamento dos servidores, tanto para sua configuração, quanto para o gerenciamento dos jogadores conectados;
c) criação de um esquema de configuração de mapas mais completo, que permita a definição de mapas com formados diferentes de quadrados, áreas não transitáveis, etc;
d) otimização dos diversos processos do servidor, como gerenciamento de memória, sincronização dos *threads*, formatos das mensagens, etc.;
e) criação de versões para outras plataformas que não o Windows, utilizando tecnologias equivalentes ao IOCP para gerenciamento das conexões.

REFERÊNCIAS BIBLIOGRÁFICAS

ASSOCIATES, Computer. **AllFusion ERWin Data Modeler**. Islandia, 2006. Disponível em: <http://www.ca.com>. Acesso: em 9 nov. 2006.

BOGOJEVIC, Sladjan. **The architecture of massive multiplayer online games**. 2003. 107 f. Dissertação de Mestrado (Mestrado em Ciências da Computação) - Department of Computer Science, Lund Institute of Technology, Lund. Disponível em: <http://graphics.cs.lth.se/theses/projects/mmogarch/som.pdf>. Acesso em: 1 ago. 2006.

FIEDLER, Stefan; WALLNER, Michael; WEBER, Michael. A communication architecture for massive multiplayer games. In: NET GAMES 2002, 1., 2002, Ulm. **Proceedings...** Braunschweig: Technical University of Braunschweig, 2002. p. 1-9. Disponível em: <http://medien.informatik.uni-ulm.de/forschung/publikationen/NetGames2002.pdf>. Acesso em: 1 ago. 2006.

IERASALIMSCHY, Roberto; FIGUEIREDO, Luis H.; CELES, Waldemar. **Lua**: an extensible extension language. Rio de Janeiro, 1996. Disponível em: <http://www.lua.org/spe.html>. Acesso em: 21 set. 2006.

IERASALIMSCHY, Roberto; FIGUEIREDO, Luis H.; CELES, Waldemar. **Lua 5.1 reference manual**. Rio de Janeiro, 2006. Disponível em: <http://www.lua.org/manual/5.1/manual.html>. Acesso em: 21 set. 2006.

IGDA - INTERNATIONAL GAME DEVELOPERS ASSOCIATION. **IGDA online games white paper**. [S.l.], 2003. Disponível em: <http://www.igda.org/online/IGDA_Online_Games_Whitepaper_2003.pdf>. Acesso em: 2 ago. 2006.

JONES, Anthony; DESHPANTE Amol. **Windows Sockets 2.0**: write scalable winsock apps using completion ports. [S.l], 1999. Disponível em: <http://msdn.microsoft.com/msdnmag/issues/1000/winsock/>. Acesso em: 21 set. 2006.

KENSHO. **MySQL C++ Wrapper**. [S.l.], 2006. Disponível em: <http://www.codeproject.com/database/mysqlwrap.asp>. Acesso em: 9 nov. 2006.

LEVEL UP! INTERACTIVE. **Level Up! games chega ao Brasil com Ragnarok, um dos maiores jogos on-line do mundo com mais de 3 milhões de jogadores em 13 países**. São Paulo, 2004. Disponível em: <http://www.ragnarok.com.br/arquivos_sala_imprensa/down.php?arquivo=351.doc>. Acesso em: 3 ago. 2006.

LYRA STUDIOS. **Lyra network engine features and architecture**. Los Angeles, 2003. Disponível em: <http://www.lyrastudios.com/assets/lyraarchitecture.pdf>. Acesso em: 3 ago. 2006.

MICROSOFT CORPORATION. **Win32 and COM Development**. Redmond, [2006?]. Disponível em: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/anch_win32com.asp>. Acesso em: 21 set. 2005.

RPG jogo. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/RPG_(jogo)>. Acesso em: 21 set. 2006.

SANTOS, André L. S. C. **VIX**: um framework para suporte a objetos visuais interativos. 1996. 86 f. Dissertação (Mestrado em Ciências da Computação) - Departamento de Ciências da Computação, PUC-Rio, Rio de Janeiro. Disponível em: <http://www.tecgraf.puc-rio.br/publications/diss_1996_clinio_vix.pdf>. Acesso em: 3 ago. 2006.

SIKORA, Drew. **Programming with asynchronous sockets**. [S.l.], 2001. Disponível em: <http://www.gamedev.net/reference/articles/article1297.asp>. Acesso em: 21 set. 2006.

SPARX, Systems. **Enterprise Architect.** Victoria, 2006. Disponível em: <http://www.sparxsystems.com.au>. Acesso: em 9 nov. 2006.

THOR, Alexander (Ed.). **Massively multiplayer game development**. 2nd ed. Massachusetts: Charles River Media Inc. 2005.

TREGLIA, Dante (Ed.). **Game programming gems 3**. 3rd ed. Massachusetts: Charles River Media Inc. 2002.

APÊNDICE A – Referência de mensagens do motor

Neste apêndice são descritas em detalhes todas as mensagens possíveis no servidor, como pode ser visto no quadro 32. Neste quadro é possível ver o nome da mensagem, sua descrição, sua direção (indicando quais as camadas de origem e destino possíveis), além de seus campos detalhados.

MSG_0000		
Descrição	Indicador de ACK, retornado pelos servidores quando uma mensagem é recebida com sucesso. Atualmente, utilizada como resposta instantânea do servidor à solicitações de movimentação de personagens, já que as mensagens de movimentação possuem um atraso para começarem a ser enviadas.	
Direção	Sessão para cliente	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0000).
msgOk	Int 16 bits	Código da mensagem que está sendo confirmada.
MSG_0001		
Descrição	Solicitação de assinatura de canal de interação.	
Direção	Sessão para localização.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0001).
idCanal	Int 32 bits	Código do canal a ser assinado.
MSG_0002		
Descrição	Solicitação de assinatura de canal de ambiente.	
Direção	Sessão para localização.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0002).
idCanal	Int 32 bits	Código do canal a ser assinado.
MSG_0003		
Descrição	Solicita cancelamento de assinatura de um canal de interação.	
Direção	Sessão para Localização	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0003).
idCanal	Int 32 bits	Canal cuja assinatura deve ser cancelada.
MSG_0004		
Descrição	Solicita cancelamento de assinatura de um canal de ambiente.	
Direção	Sessão para localização.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0004).
idCanal	Int 32 bits	Canal cuja assinatura deve ser cancelada.
MSG_0005		
Descrição	Adiciona jogador ao servidor. Antes do servidor de sessão iniciar o envio de mensagens em nome de um jogador, é necessário adiciona-lo ao servidor de localização através desta mensagem.	
Direção	Sessão para localização.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0005).
jogadorId	Int 32 bits	Código do jogador a ser adicionado ao servidor.

MSG_0006		
Descrição	Remove jogador do servidor. Ao sair de um determinado mapa ou desconectar-se do servidor de sessão, é necessário que o jogador seja removido do servidor de localização através desta mensagem.	
Direção	Sessão para localização.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0006).
jogadorId	Int 32 bits	Código do jogador a ser removido do servidor.
MSG_0007		
Descrição	Personagem apareceu. Enviada quando um personagem aparece em um determinado mapa/canal.	
Direção	Sessão para localização. Localização para sessão. Sessão para cliente.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0007).
jogadorId	Int 32 bits	Código do jogador que apareceu.
mapId	Int 32 bits	Código da parte de mapa onde o jogador apareceu.
tileId	Int 32 bits	Célula da parte de mapa onde o jogador apareceu.
MSG_0008		
Descrição	Personagem saiu. Enviada quando um personagem sai de um determinado mapa/canal.	
Direção	Sessão para localização. Localização para sessão. Sessão para cliente.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0008).
jogadorId	Int 32 bits	Código do jogador que saiu.
mapId	Int 32 bits	Código da parte de mapa onde o jogador se encontrava.
tileId	Int 32 bits	Célula da parte de mapa onde o jogador se encontrava.
MSG_0009		
Descrição	Movimentação de personagem.	
Direção	Sessão para localização. Localização para sessão. Sessão para cliente.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x0009).
jogadorId	Int 32 bits	Código do jogador que se movimentou.
mapId	Int 32 bits	Código da parte de mapa para onde o jogador se movimentou.
tileId	Int 32 bits	Código da célula para onde o jogador se movimentou.
MSG_000A		
Descrição	Solicitação de atualização dados de ambiente. Servidor de sessão solicita isso quando assina um canal, para ficar a par do que está acontecendo naquele momento no canal.	
Direção	Sessão para Localização.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x000A).
mapId	Int 32 bits	Código da parte de mapa (canal) que se deseja receber as informações.
jogadorId	Int 32 bits	Código do jogador que originou a necessidade pelas informações.
MSG_000B		
Descrição	Solicita atualização de dados de interação. Idem MSG_000A, porém, para um canal de interação.	
Direção	Sessão para localização.	
Campos		

Nome	Tipo	Descrição
idMsg	Int 16 bits	Código da mensagem (0x000B).
mapId	Int 32 bits	Código da parte de mapa (canal) que se deseja receber as informações.
jogadorId	Int 32 bits	Código do jogador que originou a necessidade pelas informações.
MSG_000C		
Descrição	Envio de chat..	
Direção	Sessão para localização. Localização para sessão. Sessão para cliente.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x000C).
mapId	Int 32 bits	Código da parte de mapa (canal) onde a mensagem aconteceu.
jogadorId	Int 32 bits	Código do jogador autor da mensagem.
msgSize	Int 8 bits	Tamanho da mensagem (apenas do texto do chat).
Msg	Char[100]	Mensagem propriamente dita.
MSG_1003		
Descrição	Solicitação de login.	
Direção	Cliente para sessão.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x1003).
loginName	Char[50]	Login do jogador.
loginPassword	Char[15]	Senha do jogador.
MSG_1004		
Descrição	Solicita movimentação de personagem.	
Direção	Cliente para sessão.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x1004).
mapId	Int 32 bits	Código da parte de mapa de destino desejada.
tileId	Int 32 bits	Código da célula de destino desejada.
MSG_1005		
Descrição	Envio de mensagens de chat (do cliente).	
Direção	Cliente para sessão.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x1005).
msgSize	Int 8 bits	Tamanho da mensagem (apenas do chat).
Msg	Char[100]	Mensagem propriamente dita.
MSG_1006		
Descrição	Solicitação de dados do jogador. Retorna os seus dados, como nome, etc.	
Direção	Cliente para sessão.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x1006).
jogadorId	Int 32 bits	Código do jogador que se deseja receber os dados.
MSG_1007		
Descrição	Solicita início de jogo após efetuar login.	
Direção	Cliente para sessão.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x1007).
MSG_1008		
Descrição	Retorno de dados de jogador após solicitação da mensagem `MSG_1006.`	
Direção	Sessão para cliente.	
Campos		

Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x1008).
jogadorId	Int 32 bits	Código do jogador.
Nome	Char[50]	Nome do jogador.
MSG_5000		
Descrição	Mensagem customizada. Deve ser utilizada para troca de mensagens entre o cliente e os scripts de extensão do servidor. Os servidores não a tratam, apenas a encaminham aos scripts, que deve efetuar seu tratamento. Embora oficialmente ela tenha só três campos, ela é extensível, podendo ter quantos campos (e tamanho) quanto for necessário. O tamanho total da mensagem, entretanto, deve sempre ser enviado corretamente no campo msgSize.	
Direção	Todas.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0x5000).
idSubMsg	Int 16 bits	Código da mensagem interna. Campo livre para identificar a mensagem nos scripts que fazem seu tratamento.
msgSize	Int 16 bits	Tamanho total da mensagem (incluindo campos extra).
MSG_FFFF		
Descrição	Retorno NACK. Enviado quando uma solicitação de movimentação de personagem do cliente não é aceita pelo servidor de localização.	
Direção	Sessão para cliente.	
Campos		
Nome	**Tipo**	**Descrição**
idMsg	Int 16 bits	Código da mensagem (0xFFFF).
errorId	Int 16 bits	Código do erro. Verificar apêndice B para códigos de erros.

Quadro 32 – Mensagens da aplicação

APÊNDICE B – Referência de códigos de erros do servidor

Neste apêndice são expostos os códigos de erros que podem ser retornados pela mensagem `MSG_FFFF`, bem como seus respectivos significados. Estes códigos estão visíveis no quadro 33.

Nome	Código	Descrição
ERR_JOGADOR_EXISTE	0x0001	Tentativa de conectar jogador que já encontra-se conectado.
ERR_JOGADOR_INVALIDO	0x0002	Referência a um jogador que não encontra-se conectado.
ERR_CANAL_INVALIDO	0x0003	Referência a uma parte de mapa (canal) que não existe.
ERR_POSICAO_INVALIDO	0x0004	Referência a uma posição no mundo virtual que não existe.
ERR_MSG_INESPERADA	0x0005	Mensagem inesperada no momento em que foi enviada.
ERR_USUARIO_INVALIDO	0x0006	Referência a usuário inválido.

Quadro 33 – Códigos de erro da aplicação

APÊNDICE C – Referência de funções de comunicação dos *Scripts* com a aplicação

Este apêndice contém a listagem das funções disponíveis para que os *scripts* de extensão Lua comuniquem-se com as camadas do servidor. São descritas as funções, seu objetivo, seus parâmetros e seus resultados (caso existam). Há também a função concreta, que é a implementação na aplicação da função descrita.

Devido à estrutura da linguagem Lua, é possível ter mais de um resultado para uma função. Além disso, ela também conta com um tipo de dados `lua_userdata`, que representa um ponteiro para a linguagem C. Não é possível acessar variáveis deste tipo de dado diretamente nos *scripts* Lua. Apenas as funções de comunicação com a aplicação conseguem acessar estes ponteiros, que normalmente representam alguma estrutura ou objeto da própria aplicação.

O quadro 34 contém as funções comuns às duas camadas. Já os quadros 35 e 36 possuem, respectivamente, as funções específicas à camada de localização e sessão.

`mysqlQuery`		
Descrição	Executa uma sentença DQL no banco de dados do servidor.	
Função Concreta	CBaseLuaScript::mysqlQuery_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Sql	String	Sentença SQL a ser executada.
Resultados		
Tipo	**Descrição**	
Integer	Número de linhas que a sentença resultou.	
Lua_userdata	Ponteiro para uma estrutura com os resultados da sentença.	
`mysqlExecute`		
Descrição	Executa uma sentença DML ou DDL (que não possua resultados) no banco de dados do servidor.	
Função Concreta	CBaseLuaScript::mysqlExecute_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Sql	String	Sentença SQL a ser executada.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`mysqlFetch`		
Descrição	Itera entre os resultados de uma sentença DQL executada através da função *mysqlQuery*.	
Função Concreta	CBaseLuaScript::mysqlFetch_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
mysqlData	Lua_userdata	Ponteiro para uma estrutura com os resultados de uma sentença, retornado pela função *mysqlQuery*.
Resultados		
Tipo	**Descrição**	
Lua_userdata	O mesmo ponteiro recebido como parâmetro.	
`mysqlGetInt`		
Descrição	Retorna o valor inteiro de uma coluna do resultado de uma sentença SQL executada através da função *mysqlQuery*.	
Função Concreta	CBaseLuaScript::mysqlGetInt(lua_State* state)	

Parâmetros		
Nome	**Tipo**	**Descrição**
mysqlData	Lua_userdata	Ponteiro para uma estrutura com os resultados de uma sentença, retornado pela função *mysqlQuery*.
colInd	Integer	Índice da coluna que se deseja obter o resultado, iniciando em zero.
Resultados		
Tipo	**Descrição**	
Integer	Valor da coluna solicitada. Zero caso a coluna referenciada não exista ou não seja do tipo inteiro.	
`mysqlGetString`		
Descrição	Retorna o valor string de uma coluna do resultado de uma sentença SQL executada através da função *mysqlQuery*.	
Função Concreta	CBaseLuaScript::mysqlGetString_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
mysqlData	Lua_userdata	Ponteiro para uma estrutura com os resultados de uma sentença, retornado pela função *mysqlQuery*.
colInd	Integer	Índice da coluna que se deseja obter o resultado, iniciando em zero.
Resultados		
Tipo	**Descrição**	
String	Valor da coluna solicitada.	
`mysqlFree`		
Descrição	Libera a memória alocada para uma estrutura de resultados de uma sentença executada através da função *mysqlQuery*. É necessário liberar a memória após finalizar a utilização destas estruturas.	
Função Concreta	CBaseLuaScript::mysqlFree_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
mysqlData	Lua_userdata	Ponteiro para a estrutura a ser destruída.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`alocaBuffer`		
Descrição	Aloca um buffer com o tamanho especificado, para utilização posterior em rotina de envio de mensagens através dos scripts de extensão. Este buffer contém, além do espaço de memória com o tamanho solicitado, um ponteiro para o empacotamento dos dados. Este ponteiro inicia com zero, e é atualizado conforme os dados vão sendo empacotados.	
Função Concreta	CBaseLuaScript::alocaBuffer_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Tamanho	Integer	Tamanho do buffer a ser alocado, em bytes.
Resultados		
Tipo	**Descrição**	
Lua_userdata	Ponteiro para o buffer alocado.	
`freeBuffer`		
Descrição	Libera um buffer alocado através da função *alocaBuffer*. Após terminar de utilizar um buffer destes, é necessário liberá-lo com esta função.	
Função Concreta	CBaseLuaScript::freeBuffer(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Buffer	Lua_userdata	Ponteiro para o buffer que se deseja liberar (retornado pela função *alocaBuffer*).
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`setPackIdx`		
Descrição	Atualiza a posição do ponteiro de empacotamento de um buffer retornado pela função *alocaBuffer*. Após atualizar a posição, as próximas chamadas para funções de	

	empacotamento ou desempacotamento de dados neste buffer acontecerão a partir da posição indicada.	
Função Concreta	CBaseLuaScript::setPackIdx_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Buffer	Lua_userdata	Buffer previamente alocado com a função *alocaBuffer*.
packIdx	Integer	Nova posição para o ponteiro de empacotamento do buffer.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`packInt`		
Descrição	Empacota um valor inteiro num buffer alocado através da função *alocaBuffer*. O valor é empacotado na posição indicada pelo ponteiro de empacotamento do buffer, e após isso adiciona ao ponteiro o tamanho de um integer (4 bytes), deixando o buffer pronto para o próximo empacotamento.	
Função Concreta	CBaseLuaScript::packInt_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Buffer	Lua_userdata	Buffer previamente alocado com a função *alocaBuffer* onde se deseja empacotar o valor.
vlInteger	Integer	Valor que se deseja empacotar no buffer.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`packChar`		
Descrição	Empacota um valor char (1 byte) num buffer alocado através da função *alocaBuffer*. O valor é empacotado na posição indicada pelo ponteiro de empacotamento do buffer, e após isso, adiciona ao ponteiro o tamanho de um integer (1 byte), deixando o buffer pronto para o próximo empacotamento.	
Função Concreta	CBaseLuaScript::packChar_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Buffer	Lua_userdata	Buffer previamente alocado com a função *alocaBuffer* onde se deseja empacotar o valor.
vlChar	Char	Valor que se deseja empacotar no buffer.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`packShortInt`		
Descrição	Empacota um valor short int (2 bytes) num buffer alocado através da função *alocaBuffer*. O valor é empacotado na posição indicada pelo ponteiro de empacotamento do buffer, e após isso, adiciona ao ponteiro o tamanho de um short int (2 bytes), deixando o buffer pronto para o próximo empacotamento.	
Função Concreta	CBaseLuaScript::packShortInt_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Buffer	Lua_userdata	Buffer previamente alocado com a função *alocaBuffer* onde se deseja empacotar o valor.
vlShortInt	Short Int	Valor que se deseja empacotar.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`unpackInt`		
Descrição	Desempacota um valor integer de um buffer criado com a função *alocaBuffer*. O valor é carregado baseando-se na posição atual do ponteiro de empacotamento do buffer. Ao final do processo, o ponteiro é decrementado com o tamanho de um integer (4 bytes), deixando o buffer pronto para outro desempacotamento.	
Função Concreta	CBaseLuaScript::unpackInt_glue(lua_State* state)	
Parâmetros		

Nome	Tipo	Descrição
Buffer	Lua_userdata	Buffer previamente alocado pela função *alocaBuffer*, e que tenha algum valor empacotado (através das funções de empacotamento disponíveis).
Resultados		
Tipo	**Descrição**	
Integer	Valor desempacotado do buffer.	
`unpackChar`		
Descrição	Desempacota um valor char de um buffer criado com a função *alocaBuffer*. O valor é carregado baseando-se na posição atual do ponteiro de empacotamento do buffer. Ao final do processo, o ponteiro é decrementado com o tamanho de um integer (1 byte), deixando o buffer pronto para outro desempacotamento.	
Função Concreta	CBaseLuaScript::unpackChar_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Buffer	Lua_userdata	Buffer previamente alocado pela função *alocaBuffer* e que tenha algum valor empacotado (através das funções de empacotamento disponíveis).
Resultados		
Tipo	**Descrição**	
Char	Valor desempacotado do buffer.	
`unpackShortInt`		
Descrição	Desempacota um valor short int de um buffer criado com a função *alocaBuffer*. O valor é carregado baseando-se na posição atual do ponteiro de empacotamento do buffer. Ao final do processo, o ponteiro é decrementado com o tamanho de um short int (2 byte), deixando o buffer pronto para outro desempacotamento.	
Função Concreta	CBaseLuaScript::unpackShortInt_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Buffer	Lua_userdata	Buffer previamente alocado pela função *alocaBuffer* e que tenha algum valor empacotado (através das funções de empacotamento disponíveis).
Resultados		
Tipo	**Descrição**	
Short int	Valor desempacotado do buffer.	
`buffer_unpack`		
Descrição	Desempacota um buffer conforme os parâmetros. Esta função possui parâmetros variáveis, podendo ser passados nas quantidades que forem necessários. Com ela é possível desempacotar diversos valores em uma única chamada de função.	
Função Concreta	CBaseLuaScript::buffer_unpack_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Buffer	lua_userdata	Buffer de onde se deseja desempacotar os valores, previamente alocado através da função *alocaBuffer*.
Tipo1	Integer	Primeiro tipo a ser desempacotado. Os valores aceitos são 0, 1 ou 2 e representam, respectivamente, integer, short int e char.
Tipo2	Integer	Segundo tipo a ser desempacotado. Os valores são iguais aos do parâmetro anterior.
...	...	...
TipoN	Integer	Enésimo tipo a ser desempacotado. Os valores são iguais aos do parâmetro anterior
Resultados		
Tipo	**Descrição**	
Conforme parâmetros	Os resultados da função são conforme os parâmetros passados a ela. Vai existir um resultado para cada um dos tipos passados a partir do segundo parâmetro. Os tipos dos resultados também serão equivalentes: O primeiro resultado terá o tipo definido no segundo parâmetro, o segundo resultado terá o tipo definido no terceiro parâmetro, etc.	
`constroiAcao`		
Descrição	Constrói uma ação para a camada em questão. Esta ação é então adicionada à fila de ações	

	do servidor.	
Função Concreta	CBaseLuaScript::constroiAcao_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Cnx	Lua_userdata	Conexão à qual se deseja atribuir a ação (é equivalente a receber uma mensagem com a ação da conexão em questão). Esta conexão é recebida de funções como *getLocConnection* (ver tabela 36).
subMsg	Integer	Código da sub-mensagem (utilizada nas mensagens customizadas 0x5000)
Buffer	Lua_userdata	Buffer com os dados a ser enviados, alocado através da função *alocaBuffer* e com os dados devidamente empacotados pelas funções de empacotamento.
vlTimer	Integer	Caso a ação deva ser agendada, deve-se passar o tempo em milissegundos que o servidor deve aguardar antes de executá-la.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`enviaMensagemDireta`		
Descrição	Envia uma mensagem diretamente a um destinatário, representado por uma conexão.	
Função Concreta	CBaseLuaScript::enviaMensagemDireta_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Cnx	Lua_userdata	Conexão à qual se deseja enviar a mensagem. Esta conexão é obtida através de funções como *getLocConnection* (ver tabela 36).
Buffer	Lua_userdata	Buffer com os dados a ser enviados, alocado através da função *alocaBuffer*, e com os dados devidamente empacotados pelas funções de empacotamento.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`print_out`		
Descrição	Imprime uma mensagem no console da aplicação.	
Função Concreta	CBaseLuaScript::print_out_glue(lua_State* lua)	
Parâmetros		
Nome	**Tipo**	**Descrição**
Texto	String	Texto a ser impresso.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	

Quadro 34 – Funções de comunicação com *scripts* Lua comuns às camadas

`getPosicaoInfo`		
Descrição	Retorna os dados de posição de um determinado jogador.	
Função Concreta	CLocScript::getPosicaoInfo_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
scriptObj	Lua_userdata	Referência ao objeto de scripts em execução. Ele é passado ao ambiente Lua no evento EVT_OnIniciaServer, e deve ser armazenado para ser usado com esta e outras funções.
playerId	Integer	Código do jogador que se deseja receber as informações.
Resultados		
Tipo	**Descrição**	
Integer	Código da parte de mapa (canal) onde o jogador se encontra.	
Integer	Coordenada X em que o jogador se encontra na parte do mapa.	
Integer	Coordenada Y em que o jogador se econtra na parte do mapa.	
`getConnection`		
Descrição	Retorna o objeto de conexão com o servidor de sessão de um determinado jogador, que pode ser utilizado pelas funções que enviam mensagens.	
Função Concreta	CLocScript:: getConnection_glue(lua_State* state)	

Parâmetros		
Nome	**Tipo**	**Descrição**
ScriptObj	Lua_userdata	Referência ao objeto de scripts em execução. Ele é passado ao ambiente Lua no evento EVT_OnIniciaServer, e deve ser armazenado para ser usado com esta e outras funções.
playerId	Integer	Código do jogador que se deseja obter a conexão.
Resultados		
Tipo	**Descrição**	
Lua_userdata	Referência ao objeto de conexão do jogador com o servidor de sessão.	
`enviaIntAssinantes`		
Descrição	Envia uma mensagem para os assinantes de um canal de interação.	
Função Concreta	CLocScripts::enviaIntAssinantes_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
scriptObj	Lua_userdata	Referência ao objeto de scripts em execução. Ele é passado ao ambiente Lua no evento EVT_OnIniciaServer, e deve ser armazenado para ser usado com esta e outras funções.
canalId	Integer	Código do canal a que se deseja enviar a mensagem.
buffer	Lua_userdata	Buffer criado pela função *alocaBuffer* com os dados que deseja-se enviar devidamente empacotas pelas funções de empacotamento disponíveis.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`enviaAmbAssinantes`		
Descrição	Envia uma mensagem para os assinantes de um canal de ambiente.	
Função Concreta	CLocScripts::enviaAmbAssinantes_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
scriptObj	Lua_userdata	Referência ao objeto de scripts em execução. Ele é passado ao ambiente Lua no evento EVT_OnIniciaServer, e deve ser armazenado para ser usado com esta e outras funções.
canalId	Integer	Código do canal a que se deseja enviar a mensagem.
buffer	Lua_userdata	Buffer criado pela função *alocaBuffer* com os dados que deseja-se enviar devidamente empacotas pelas funções de empacotamento disponíveis.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	

Quadro 35 – Funções de comunicação com scripts Lua da camada de localização

`enviaMensagem`		
Descrição	Envia uma mensagem a um jogador.	
Função concreta	CSesScripts::enviaMensagem_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
scriptObj	Lua_userdata	Referência ao objeto de scripts em execução. Ele é passado ao ambiente Lua no evento EVT_OnIniciaServer, e deve ser armazenado para ser usado com esta e outras funções.
msgId	Integer	Código da mensagem a ser enviada
buffer	Lua_userdata	Buffer previamente criado através da função *alocaBuffer* e com os dados a serem enviados devidamente empacotados através das funções de empacotamento disponíveis.
playerId	Integer	Código do jogador a quem se deseja enviar a mensagem.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`enviaMensagemInteracao`		
Descrição	Envia uma mensagem a um canal de interação.	

Função concreta	CSesScripts::enviaMensagemInteracao_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
scriptObj	Lua_userdata	Referência ao objeto de scripts em execução. Ele é passado ao ambiente Lua no evento EVT_OnIniciaServer, e deve ser armazenado para ser usado com esta e outras funções.
msgId	Integer	Código da mensagem a ser enviada.
buffer	Lua_userdata	Buffer previamente criado através da função *alocaBuffer* e com os dados a serem enviados devidamente empacotados através das funções de empacotamento disponíveis.
CanalId	Integer	Código do canal ao qual se deseja enviar a mensagem.
Resultados		
Tipo	**Descrição**	
<nenhum>	<nenhum>	
`getLocConnection`		
Descrição	Retorna o objeto de conexão ao servidor de localização em que um determinado jogador se encontra.	
Função concreta	CSesScripts::getLocConnection_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
scriptObj	Lua_userdata	Referência ao objeto de scripts em execução. Ele é passado ao ambiente Lua no evento EVT_OnIniciaServer, e deve ser armazenado para ser usado com esta e outras funções.
playerId	Integer	Código do jogador de quem se deseja obter o objeto de conexão.
Resultados		
Tipo	**Descrição**	
Lua_userdata	Objeto de conexão ao servidor de localização em que o jogador se encontra.	
`getClientConnection`		
Descrição	Retorna o objeto de conexão com um determinado jogador propriamente dito.	
Função concreta	CSesScripts:: getClientConnection_glue(lua_State* state)	
Parâmetros		
Nome	**Tipo**	**Descrição**
scriptObj	Lua_userdata	Referência ao objeto de scripts em execução. Ele é passado ao ambiente Lua no evento EVT_OnIniciaServer, e deve ser armazenado para ser usado com esta e outras funções.
playerId	Integer	Código do jogador a quem se deseja enviar a mensagem.
Resultados		
Tipo	**Descrição**	
Lua_userdata	Objeto de conexão direta ao jogador.	

Quadro 36 – Funções de comunicação com *scripts* Lua da camada de sessão