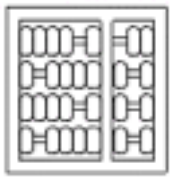

INSTITUTO DE COMPUTAÇÃO

UNIVERSIDADE ESTADUAL DE CAMPINAS

AlphaMMO – Servidor para

Jogos Multijogador

V. Roth Cardoso *A. Santanchè*

Relatório Técnico - IC-PFG-16-15 - Proieto Final de Graduacão

December - 2016 - Dezembro

AlphaMMO – Servidor para Jogos Multijogador

Victor Roth Cardoso[1], André Santanchè[1]

[1] Instituto de Computação Universidade Estadual de Campinas (UNICAMP), Caixa Postal 6176
13083-970 Campinas-SP, Brasil

victor.roth.cardoso@gmail.com

Resumo. Este relatório trata do trabalho final de graduação do aluno Victor Roth Cardoso, cujo foco é um estudo para a criação de um servidor para jogos multijogador, bem com a sua implementação. O estudo foi feito em diversas etapas: o estudo de diversos design patterns para jogos, estudo de arquiteturas para jogos multijogador e a implementação de um jogo cliente-servidor. A implementação considera as demandas para um jogo online, incluindo aspectos de performance, segurança e de engenharia de software.

Palavras-Chave: Jogos; Cliente-Servidor; Jogos multijogadores; Multiplayer; Arquitetura de software; Design Patterns; Microthreads.

1. Introdução

Jogos multijogadores, comumente chamados de jogos multiplayers, são jogos em que há a participação de, pelo menos, dois jogadores.

Há dois tipos de jogos comumente chamados multijogador: *jogo em um único sistema* e *jogos cliente-servidor*. No primeiro tipo, os dois jogadores podem jogar com a tela compartilhada a todo momento como no caso dos jogos *Worms* e *Minecraft* em console de videogames (respectivamente disponíveis em: https://www.team17.com/games/ e https://www.playstation.com/en-us/games/minecraft-ps4/). Alternativamente, nos jogos de um único sistema os jogadores podem compartilhar a mesma tela em momentos distintos, como no caso de jogos como da série de jogos *Civilization* e *jogos via redes* (Civilization está disponível em https://www.civilization.com/). Nos jogos do tipo cliente-servidor, há a necessidade de um *servidor* para prover os serviços do jogo e os usuários jogam em terminais distintos, *clientes* – este é o caso de jogos como *Ultima Online*, *Tibia*, *Runescape* e *World of Warcraft* (disponíveis em: http://uo.com/, http://www.tibia.com/, http://www.runescape.com/ e https://worldofwarcraft.com/).

Dentro de cada um destes tipos citados acima há também gêneros diferentes que definem objetivos ou formas de iterações diferentes [WIK2, 2016].

Alguns jogos estão em mais de uma categoria, tanto no gênero, como nos tipos citados acima. Por exemplo, nos jogos da série *Civilization*, o jogador pode alternar o modo de jogo, de forma que o jogo tipicamente jogado em modo multijogador com tela compartilhada pode ser jogado através de um servidor remoto.

Neste trabalho foi investigada a criação de um jogo multijogador considerando a modalidade de *jogo cliente-servidor via rede*, com foco no estudo de jogos MMO, *Massive Multiplayer Online.* Nestes jogos há um universo dinâmico em que os jogadores interagem. Estes jogadores entram e saem do jogo sem nenhuma restrição de início de partida [GSA, 2015].

2. Justificativa

Este trabalho foi realizado devido ao crescente número de jogos multijogador. É uma área com diversos desafios computacionais e também com grande potencial de mercado [WCP, 2015].

Há vários motivos para o crescimento do número de jogos multijogador. Do ponto de vista do acesso, há mais pessoas utilizando dispositivos móveis, como celulares e tablets com acesso à internet. Computadores de mesa normalmente estão conectados na internet e cada vez mais no Brasil. O último relatório da Cetic indica que 51% dos domicílios no Brasil têm acesso à internet [CET, 2015]. Há cada vez mais serviços de venda e distribuição de jogos online como *Steam* e *Nuuvem* (disponíveis em: http://store.steampowered.com/ e http://www.nuuvem.com/). Há a influência de avanços em tecnologia, visto tanto por causa de serviços online, como *Amazon Web Services* (disponível em https://aws.amazon.com) com facilidades para a criação de serviços online, quanto por conta de programas de computador para desenvolvimento de jogos, como *Unity* e *GameMaker*, aplicativos que facilitam a criação de jogos (disponíveis respectivamente em: https://unity3d.com/pt/ e http://www.yoyogames.com/gamemaker).

Uma consequência do aumento do número de jogadores é o surgimento de competições nos jogos multijogador visto em programas novos de TV a cabo sobre *e-Sports* (*Eletronic Sports* – esportes eletrônicos), onde as partidas de jogos como *League of Legends* e *Counter Strike* são narradas em tempo real na ESPN (informações sobre competições estão disponíveis em https://gamersclub.com.br/campeonatos e http://espn.uol.com.br/tag/counterstrike; os jogos citados estão disponíveis em: http://www.leagueoflegends.com e http://store.steampowered.com/app/10/).

3. Objetivo

Implementar um servidor para jogos multijogador baseado no estudo e análise de arquiteturas existentes.

4. Análise e desenvolvimento

Primeiro, descreveremos pontos a serem levados em consideração para o desenvolvimento de um jogo, observando os elementos que o compõem. Seguiremos com o estudo de questões para o desenvolvimento de uma *Engine* – estrutura básica sobre a qual o jogo é construído – considerando o caso de um jogo individual e suas modificações para um jogo multijogador simples [GRE, 2009]. Depois serão abordados alguns tópicos relacionados ao desenvolvimento de jogos em geral, tratando de questões de Engenharia de Software, como *Design Patterns,* com foco em jogos e suas modificações, inspirados nos padrões clássicos publicados por Gamma et al. [GOF, 1994; NYS, 2014]. Por fim será abordada a arquitetura de jogos MMO seguida da explicação sobre as escolhas e a experiência no desenvolvimento do projeto.

4.1. Elementos de um jogo

De maneira geral, em um jogo há diversos elementos: o jogador, o ambiente e regras [WIK, 2016]. O jogador controla uma entidade, um personagem, o qual interage com o ambiente através das regras. Tais regras são as leis deste universo, ou o conjunto de ações que o jogador consegue realizar dentro do ambiente do jogo com o seu personagem. O ambiente é formado pelo mapa, por entidades que o jogador não controla NPCs (*non-playable-characters, jogadores-não-controlados*) que são jogadores que são controlados pelo computador e entidades controladas por outros jogadores.

Para o nosso jogo temos:

- Entidade: guardará informações de seres dinâmicos no jogo: *jogadores controlados* e *jogadores-não-controlados*; e informações de efeitos visuais na tela que serão

entidades momentâneas.

- Ambiente do jogo: contém as informações sobre o mapa do jogo, i.e., a imagem de cada ponto dele; informações sobre a possibilidade de se mover para cada um destes pontos; e a localização de entidades.
- Regras: o jogador poderá executar comandos pelo teclado, para a movimentação e o aparecimento de efeitos na tela usando as teclas direcionais e a barra de espaço, limitado pela possibilidade de andar para o destino e de executar os efeitos, magias.

4.2. *Engine*

A *Engine*, o motor sobre o qual o jogo é construído, é a peça fundamental. No livro *Game Engine Architecture* de Jason Gregory são descritos componentes e detalhes práticos para o seu desenvolvimento [GRE, 2009].

Um jogo precisa no mínimo de 3 partes para o seu funcionamento: um componente que desenha na tela (saída dos dados), outro para a entrada de comandos do jogador (entrada dos dados) e um último para executar as regras e guardar os estados dos usuários (máquina de estados). No caso de um jogo multijogador dividimos os estados dos usuários em duas partes: a parte que o servidor sabe – i.e., a versão oficial do que está acontecendo no jogo – e uma cópia para cada usuário. Para cada entrada que um jogador faz, o seu comando é validado no servidor e a respectiva alteração de estado é retornada para todos os jogadores próximos. Tais jogadores redesenharão a sua visão do estado do jogo.

No nosso jogo fazemos uma abordagem *Model-View-Controller* com a passagem de mensagens entre componentes que atuam independentemente [WIK3, 2016]. Adotamos esta aproximação por questões de simplicidade no projeto.

4.3. Design Patterns

Citamos abaixo vários *design patterns* que foram utilizados no desenvolvimento do projeto. No livro *Game Programming Patterns* de Robert Nystrom são abordados diversos padrões comumente utilizados no desenvolvimento de jogos [NYS, 2014]. Há alguns *Padrões mapeados* para jogos dos padrões da literatura famosa de Gamma et al. [GOF, 1994] e outros padrões que são comumente utilizados em jogos, *Padrões de sequenciamento* e *Padrões de desacoplamento*.

4.3.1. Padrões mapeados

- *Flyweight*: este padrão tem foco no compartilhamento entre objetos do máximo de informações possíveis referentes ao estado de cada objeto, para evitar um desperdício de memória, por conta do uso de valores iguais para objetos de uma mesma classe. Na perspectiva de jogos, o objetivo deste padrão é compactar a memória dos objetos que podem ter dados repetidos, como no caso de informações do mapa do jogo, em que objetos que estão em locais diferentes têm comportamentos muito parecidos, se não idênticos, assim é possível de ter apenas um objeto para identificar um tipo de ponto no mapa que se repete por ele todo. Muitas vezes informações visuais de entidades no jogo são repetidas, dois *jogadores-não-controlados* não têm a necessidade de ter cópias individuais da mesma imagem.
- *Observer*: quando há necessidade avisar um objeto de um evento criamos um canal entre eles, de forma que toda vez que ocorre algum evento o observado envia uma mensagem para todos os observadores. Em um jogo, isto é aplicado quando um usuário se move, fala alguma coisa ou interage de qualquer maneira com o ambiente e todos a sua volta devem perceber as suas ações.
- *Prototype*: utilizado para definir um padrão comum em um objeto. Este padrão é utilizado para facilitar a criação de outros objetos pela clonagem deste objeto modelo. Em jogos, este pattern é utilizado para definir padrões de *jogadores-não-controlados*, já que os inimigos instanciados terão o mesmo comportamento para um mesmo objeto modelo.
- *Singleton*: serve para evitar que mais de uma instância de uma classe seja criada. Útil para o caso em que apenas um objeto deve existir. Em jogos isto normalmente ocorre para simplificar a criação de componentes da *Engine*, como também no servidor para o controle de mapa, já que o mapa deve criar apenas uma cópia de cada região dele.

4.3.2. Padrões de sequenciamento

- *Double buffer*: o desenho do estado do jogo para a visualização por um jogador é muito custoso computacionalmente, devido à quantidade de elementos visuais que devem ser desenhados por diversas funções e a transferência desta informação para

a memória de vídeo. Para resolver esta questão é utilizado um *buffer* enquanto o jogo desenha e, quando o jogo acaba de desenhar o novo estado do jogo, o *buffer* é trocado, evitando que o jogador visualize desenhos parciais.

- *Game loop*: na *Engine* são executadas diversas etapas para cada passo de iteração do jogador. Temos um loop, que terá um passo de entrada de comandos do jogador, atualização do estado do jogo feita pelo simulador e resposta ao jogador, desenhando na tela.
- *Update method*: este padrão indica que há uma interface comum para os componentes que podem ser interagidos. É utilizado para que cada entidade do jogo responda à simulação efetuada a cada iteração no jogo, respondendo a cada evento que afeta ela.

4.3.3. Padrões de desacoplamento

- *Component*: separamos diferentes partes de uma entidade em componentes para tratar de coisas diferentes, como a parte de simulação do comportamento físico e a visual de uma entidade.
- *Event Queue*: num jogo com múltiplos jogadores há uma ordem para a execução dos comandos definida por quem enviou antes a mensagem, da mesma forma há uma fila de eventos que são comunicados entre cada componente do jogo.

4.4. Arquiteturas de jogos MMO

Nesta seção analisaremos brevemente duas arquiteturas para jogos multijogadores.

4.4.1. Monolítica

Neste padrão de arquitetura, um servidor mantém todos os componentes juntos em uma aplicação [RIC, 2014]. Esta forma provê alguns benefícios: temos um servidor enxuto; é mais prático para testar; e podemos transferi-lo para outras máquinas sem a necessidade de se subir outras máquinas com servidores auxiliares de banco de dados, serviços para login, firewall e controle da infraestrutura. Mantendo todos os componentes juntos num mesmo sistema, trocamos a possibilidade de aumentar a quantidade de jogadores num mesmo ambiente de jogo por uma menor complexidade no desenvolvimento (Figura 1).

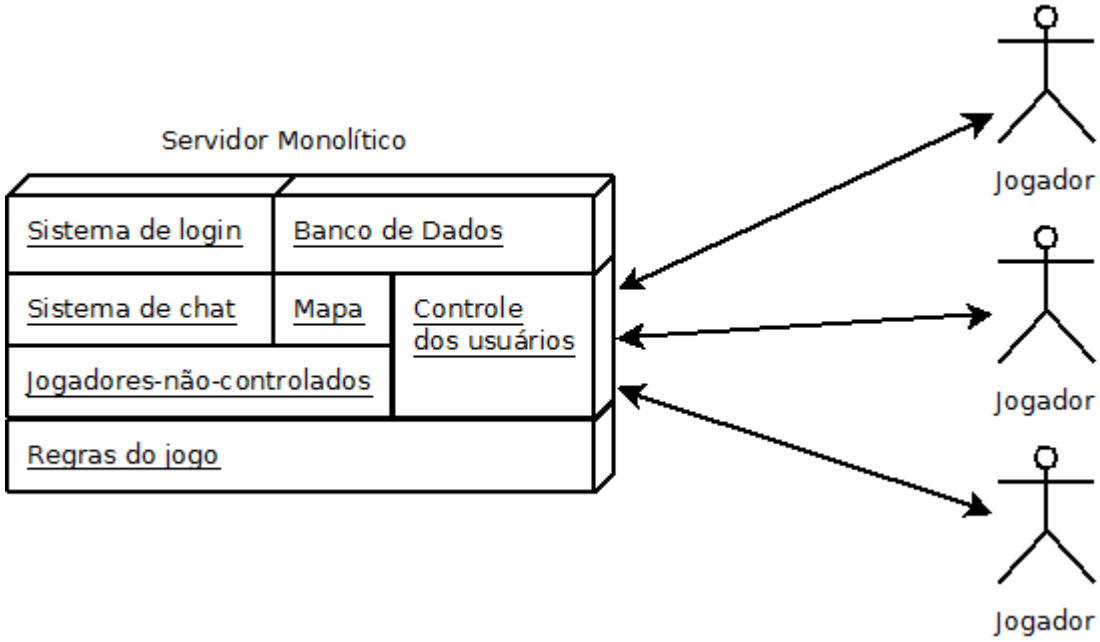

Figura1: Representação de um servidor monolítico.

4.4.2. Distribuída

Quando temos um servidor com maiores demandas de mapa e de jogadores, podemos quebrar partes do jogo em diversos servidores mantendo para o usuário uma perspectiva de continuidade. Para isto ser feito há a quebra do serviço do jogo em diversos servidores (Figura 2). Os jogadores apenas se conectarão nos servidores de fronteira (Firewall), os quais processarão a mensagem recebida e repassarão a mensagem do jogador para o servidor interno (Servidor para uma Região) com a região do mapa que o usuário se encontra no jogo. Junto aos servidores internos haverá servidores centrais de login (Servidor de login) e de banco de dados (Banco de Dados). Alguns jogos como o *Eve Online* fazem a quebra automática dos servidores de mapas em diversos outros conforme a demanda de usuários por uma região aumenta [CCP, 2007].

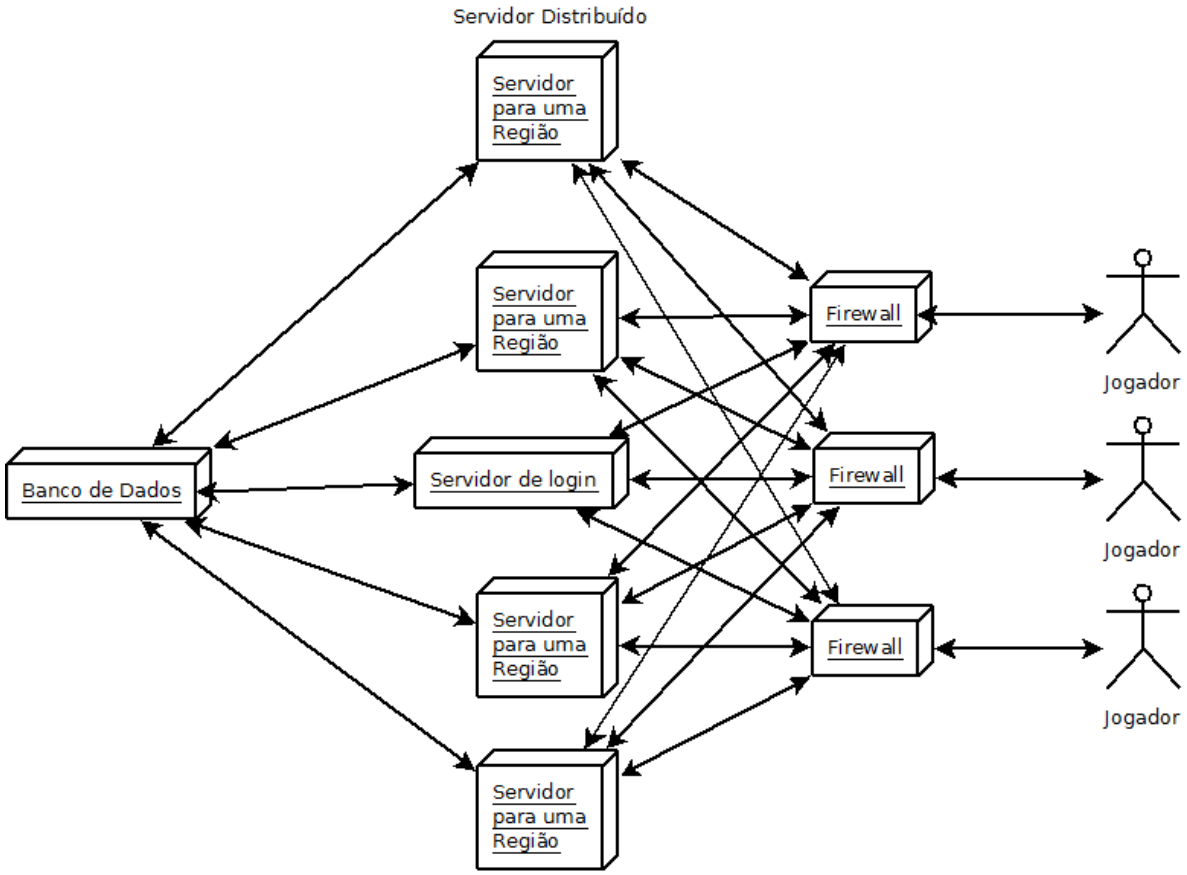

Figura 2: Representação de um servidor distribuído.

Nesta arquitetura, as diversas partes do servidor distribuído ficam separadas até fisicamente, para evitar que uma falha de um servidor comprometa o serviço todo do jogo e o sistema de banco de dados costuma conter redundância para aumentar a confiabilidade.

5. AlphaMMO – Servidor para Jogos Multijogador

5.1. Arquitetura

No desenvolvimento deste projeto escolhemos adotar a aproximação de arquitetura monolítica, separando os componentes ao máximo possível. Mostramos a seguir uma representação dos diversos componentes do nosso servidor (Figura 3).

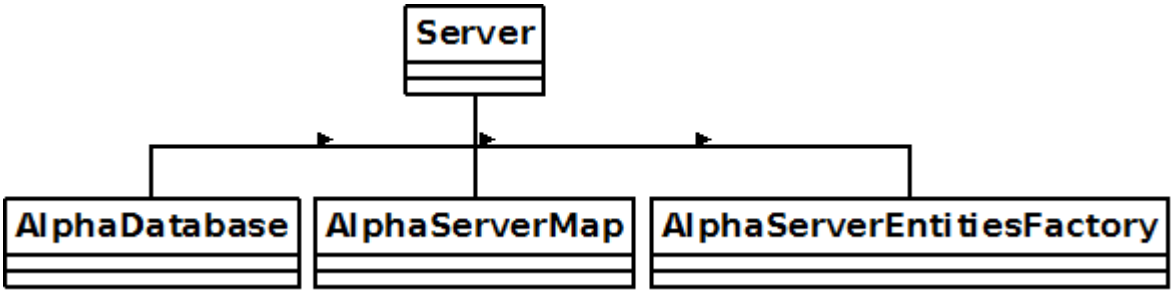

Figura 3: Representação para o nosso servidor.

Em nossa implementação o controle das entidades, tanto *jogadores-não-controlados* e *jogadores-controlados,* é feita através de um sistema de *microthreads*, em que cada entidade tem uma *microthread* (*AlphaServerEntitiesFactory* faz a criação de cada microthread para as entidades). As regras do jogo não estão em um componente específico, elas estão juntamente ao mapa em *AlphaServerMap*. As regras são gerenciadas pela *microthread* associada ao jogador e o mapa (*AlphaServerMap* e as *microthreads* de *AlphaServerEntitiesFactory*). Utilizamos um banco de dados relacional embutido em *AlphaDatabase*.

A implementação das *microthreads* é similar ao modelo enunciado no livro de Silberschatz (na seção 4.3.1, *Many-to-one Model* em *Multithreading Models)* onde as *microthreads* são gerenciadas pelo espaço de usuário, sendo assim mais leves e eficientes, porém rodando em apenas um único core [SIL, 2013].

5.2. Escolha de ferramentas

Utilizamos o *Stackless Python* por conta das *microthreads* e da disponibilidade de bibliotecas *Python* de fácil uso [STA, 2016]:

- **PyGame**: diversas funções para o desenvolvimento de jogos, como primitivas para desenho na tela, entrada de comandos pelo mouse e teclado [PYG, 2016].
- **OpenSSL**: utilizada para segurança em comunicações TCP/IP [PYO, 2016].
- **Pickle**: funcionalidade para a serialização e deserialização de objetos [PYP, 2016].

5.3. Comunicação e protocolo

Para a comunicação dos clientes com o servidor utilizamos SSL sobre sockets não-bloqueantes TCP/IP, de forma a manter segura a senha do jogador na hora do login. As mensagens trocadas são objetos serializados pelo *Pickle*, e toda vez que são deserializados os tipos de cada parte do objeto são verificados. Criamos um protocolo próprio no nível de aplicação, de forma que os canais de mensagens possam ser checados por problemas e atividades suspeitas. O protocolo funciona da seguinte forma:

- Há um grupo de comandos possíveis de serem enviados pro servidor:
    - LOGIN, LOGOUT, REGISTER, REQUEST_MOVE, REQUEST_SPEAK, REQUEST_ACTION, PING, PONG, REQUEST_RECONNECT
- E um grupo de comandos possíveis de serem enviados para os clientes:
    - STATUS, SERVER_MESSAGE, TELEPORT, MOVING, MOVING_STATUS, SPEAKING, ACTION, EFFECT, SET_ENTITIES, REPLACE_ENTITIES, ADD_ENTITIES, REMOVE_ENTITIES, SET_ENTITY, ADD_ENTITY, REMOVE_ENTITY, RECEIVE_MAP, RECEIVE_PLAYER, PING, PONG
- Toda vez que há uma mensagem para ser enviada é feito a serialização do objeto (uma tupla em Python com a sequência de dados da mensagem) e é enviado pela rede inicialmente o tamanho da mensagem, separado por um delimitador seguido da mensagem.
- A função que obtém os dados do socket guarda um buffer que acumula conforme há dados na entrada, despachando uma mensagem para processamento quando o tamanho da mensagem for atingido e a mensagem passar nas verificações de

segurança.

5.4. Banco de dados

O nosso banco de dados tem duas tabelas:

- **users_login**: contém a informação de login dos usuários – o id da conta, um código de salt (que é utilizado para toda a geração de hash juntamente da senha) e a hash da senha com o código de salt e o e-mail do jogador. A hash da senha é gerada com o código de salt e a senha em texto. Desta forma podemos garantir que se o banco de dados for comprometido não será trivial para um criminoso recuperar a senha, já que não é possível obter a senha em texto puro consultando apenas pela hash.
- **characters**: contém o nome da conta, do personagem e informações dele, como posição e data do último login.

5.5. Funcionamento do jogo

O jogo segue o seguinte ordem de execução:

1. Inicialização
    a) Inicia o mapa
    b) Inicia entidades não controladas
    c) Inicia conexões de rede
2. Espera por jogadores e entidades não controladas:
    a) Para cada jogador conectado:
        i. Recebe comandos e envia o resultado para todos os jogadores próximos
    b) Para cada entidade:
        i. Simula seu comportamento e envia o resultado para todos os jogadores próximos
3. Continue no passo 2

6. Resultados

O projeto implementado pode ser visto no Youtube acessando o link https://www.youtube.com/playlist?list=PLaFHPPA2BobVshSoUfJW_dU9SmzorwESD onde é apresentada a implementação de grandes partes do projeto. Seu código está disponível no GitHub em http://www.github.com/labrax/AlphaMMO. O jogo funciona tanto no Linux quanto no Windows, necessitando da instalação de alguns pacotes. Instruções de uso podem ser encontradas no GitHub [GIT, 2016].

Na implementação do cliente é utilizado uma *microthread* para cada um dos componentes que tratam da entrada de dados da rede, a entrada de dados do usuário e o desenho na tela.

No servidor cada jogador-não-controlado tem uma *microthread*, e, quando a entidade deixa de existir a *microthread* termina. No caso de mais jogadores-não-controladores surgirem, mais *microthreads* são criadas. Também, para cada jogador conectado há uma *microthread*, criada e terminada conforme a demanda.

6.1. Funcionamento do cliente

Explicamos nesta sessão as diversas etapas de funcionamento do nosso jogo.

6.1.1. Tela inicial

Figura 4: Tela de abertura do jogo.

No momento em que o jogador abre o jogo temos uma tela (Figura 4) com o nome do jogo e dois botões. A tela e o botão foram implementados utilizando as primitivas do *PyGame* e o comportamento é comum de janelas e seus elementos.

6.1.2. Registro

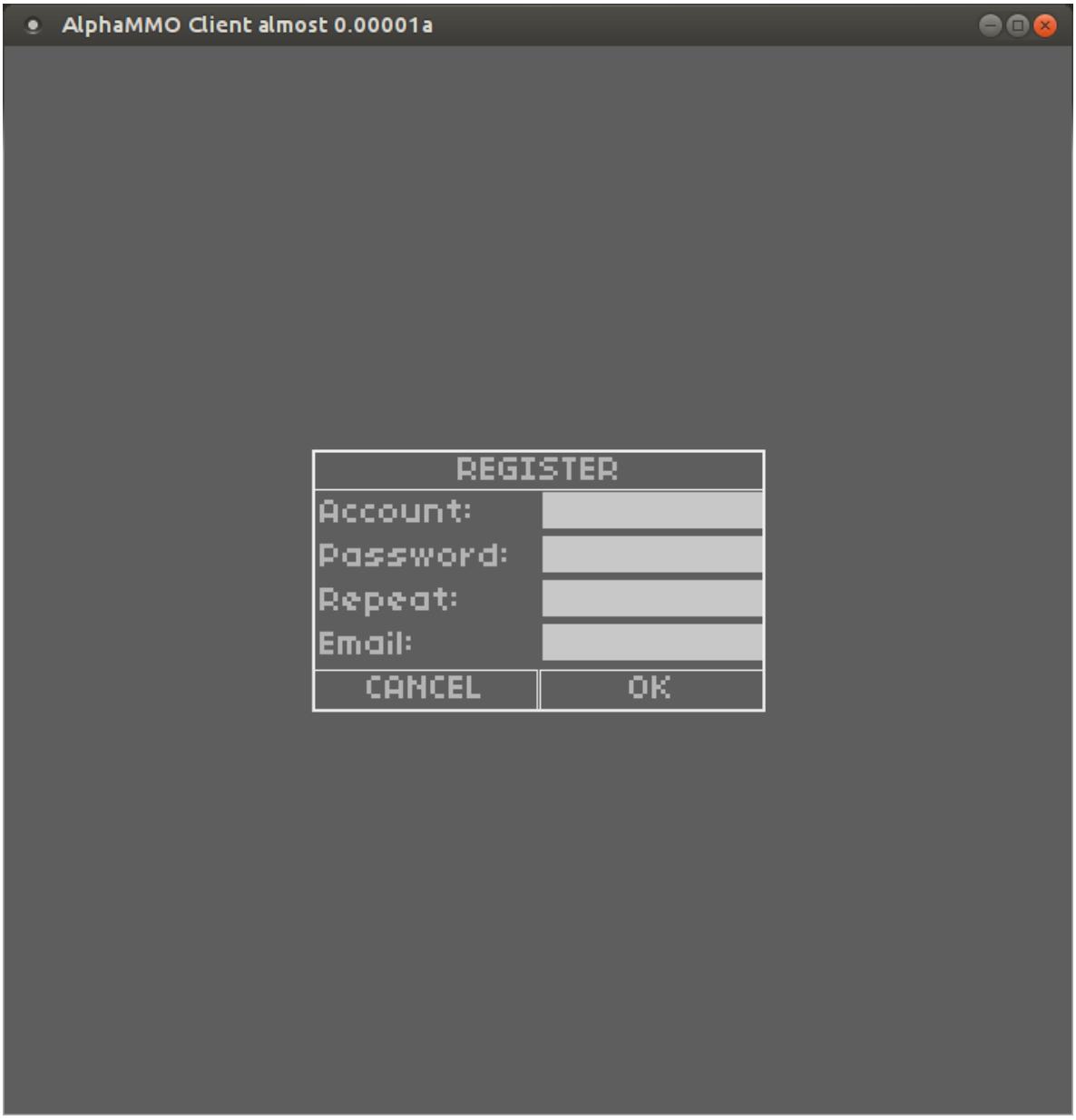

Figura 5: Tela de registro de um usuário.

Para poder acessar o jogo o usuário deve fazer um cadastro no mesmo (Figura 5). A tela de registro fornece os campos com informações para o usuário entrar. Assim que o usuário clica em *Ok* as informações são enviadas para o servidor que decidirá se é possível de criar a conta com o nome fornecido.

6.1.3. Login

Figura 6: Tela de login.

Após o usuário informar os seus dados na tela de login (Figura 6) o cliente irá se comunicar com o servidor, estabelecendo uma conexão segura e validando as informações passadas. Caso as informações estejam corretas o cliente receberá os dados do personagem dele, o mapa próximo – a região visível na tela do jogador mais uma borda de 2 quadrados para todos os lados – e as entidades vizinhas. Em seguida, ele é movido para o local que ele começará a jogar – no momento em que o cliente recebe todas estas informações ele está

pronto para começar a jogar. No mesmo momento em que este usuário se conecta as entidades em volta receberão a notificação deste usuário se conectando, e o estado do mapa no servidor mudará.

6.1.4. Jogo

Figura 7: Tela principal do jogo.

Quando o jogador começa a jogar, é feito o envio de mensagens suas para o servidor com

comandos como o de locomoção. O servidor valida os comandos do usuário, depois há a atualização dos estados enviando o estado para todos para os jogadores próximos. Na implementação feita, o jogador fica com o nome da conta, neste caso *ABC*, e com um aspecto visual aleatório dado pelo servidor. Os nomes e o visual dos *jogadores-não-controlados* são gerados aleatoriamente. O mapa foi criado manualmente pela configuração de quadrados individuais.

O jogo possui algumas regras simples: pressionar a tecla espaço faz com que o jogador faça uma magia no local que ele está, fazendo efeito visível para os demais jogadores. O jogador pode se mover pelo mapa utilizando as teclas direcionais, mas não pode ir além dos limites do mapa e se mover para cima de alguns objetos ou outros jogadores.

Durante o jogo, há uma intensa troca de mensagens com o servidor, como pode ser visualizado nas figuras a seguir, que são partes do registro de mensagens (Figuras 8 e 9):

```
('AlphaPlayMode', 'received', [<AlphaProtocol.SET_ENTITIES: 3000>, [<util.alpha_entities.Entity instance
at 0x7fdfef5a9d40>]])
('To client', [<AlphaProtocol.SET_ENTITIES: 3000>, [<util.alpha_entities.Entity instance at 0x7fdfef5a97a
0>]])
('AlphaPlayMode', 'received', [<AlphaProtocol.SET_ENTITIES: 3000>, [<util.alpha_entities.Entity instance
at 0x7fdfef5a97a0>]])
('To client', [<AlphaProtocol.SET_ENTITIES: 3000>, [<util.alpha_entities.Entity instance at 0x7fdfef5a9b4
8>]])
('AlphaPlayMode', 'received', [<AlphaProtocol.SET_ENTITIES: 3000>, [<util.alpha_entities.Entity instance
at 0x7fdfef5a9b48>]])
```

Figura 8: Trecho das mensagens recebidas pelo cliente.

Neste trecho da Figura 8 há dois tipos diferentes de mensagem, as mensagens recebidas pela interface com o servidor, identificadas pelo começo da linha escrita "*To client"*, e as mensagens recebidas pelo módulo de jogo, "*AlphaPlayMode"*.

As mensagens do servidor têm o identificador da mensagem, neste caso *AlphaProtocol.SET_ENTITIES*, código *3000* – para atualizar a informação das entidades do jogo – e como parâmetros são passadas as entidades. Ao ser recebida pela interface com o servidor a mensagem é processada e então repassada para o componente que lida com o estado do jogo. Neste caso, podemos ver várias mensagens pois há vários usuários se movendo no jogo e cada mensagem é a atualização de uma entidade.

```
('SERVER RECEIVED', [<AlphaProtocol.REQUEST_MOVE: 10>, 6, 4])
('AlphaServerPlayerTasklet', 'SERVER RAW', 10006, [<AlphaProtocol.REQUEST_MOVE: 10>, u'6', u'3'])
('SERVER RECEIVED', [<AlphaProtocol.REQUEST_MOVE: 10>, 6, 3])
('AlphaServerPlayerTasklet', 'SERVER RAW', 10006, [<AlphaProtocol.REQUEST_MOVE: 10>, u'6', u'4'])
('SERVER RECEIVED', [<AlphaProtocol.REQUEST_MOVE: 10>, 6, 4])
('AlphaServerPlayerTasklet', 'SERVER RAW', 10006, [<AlphaProtocol.REQUEST_MOVE: 10>, u'6', u'5'])
('SERVER RECEIVED', [<AlphaProtocol.REQUEST_MOVE: 10>, 6, 5])
```

Figura 9: Trecho das mensagens recebidas pelo servidor.

No caso do servidor, há o recebimento de uma mensagem pela *microthread* (aqui identificada como *AlphaServerPlayerTasklet*) do jogador, a qual processa a mensagem validando o comando do jogador e o executando retornando em seguida o resultado, como pode ser visto na Figura 8 onde há a passagem da informação das entidades visíveis no local que o jogador andou com *REQUEST_MOVE*.

6.1.5. Logout

Quando o usuário encerra o jogo ele fecha a parte de comunicação dele, fazendo com o que o servidor identifique o fim da conexão. Com isto o servidor avisa os demais jogadores que o jogador saiu, assim todos deixam de guardar informações dele e ele sai do mapa.

6.2. Inicialização do servidor

O servidor inicia e faz a seguinte sequência: cria o banco de dados, inicia o mapa, cria *jogadores-não-controlados* em *microthreads* individuais e fica esperando por conexões dos jogadores. As conexões dos jogadores são transferidas para *microthreads* e o jogo continua.

7. Conclusões e limitações

Neste trabalho alcançamos o nosso objetivo de estudar o funcionamento de um jogo multijogador, com o estudo da literatura e a implementação do jogo cliente-servidor. Nossa contribuição segue dos trabalhos realizados como objetivos, o estudo da literatura e a implementação, tanto como o estudo de problemas e soluções.

Durante o percurso deste projeto percebemos algumas limitações no projeto. Abordamos aqui algumas soluções dadas e melhorias que devem ser feitas para melhorar o projeto.

- Uso excessivo de CPU no cliente: no começo do desenvolvimento não estávamos limitando a quantidade de quadros desenhados na tela do jogador, assim desenhando o máximo que o computador permite. Isto fez com que desenhássemos quadros até

quando não há modificação na tela. Este problema foi solucionado adicionando atrasos no loop, esperando haver mais informações para se desenhar de novo.

- Uso excessivo de CPU no servidor: o uso de *microthreads* faz com que tenhamos que ficar oscilando, esperando por iterações das entidades. Caso nenhuma delas tenha operações na fila de espera, temos que verificar mesmo assim. Este loop consome muito processamento caso não coloquemos nenhuma espera, desta forma colocamos uma pequena espera para evitar a sobrecarga da máquina.

- Uso excessivo de CPU na renderização do *PyGame*: em testes realizados no Windows o *Pygame* utiliza a renderização por software no Windows, o que costuma sobrecarregar a máquina. Uma melhora para este problema é utilizar as bibliotecas de *OpenGL*, uma biblioteca para uso de funções de vídeo otimizadas que usam a GPU para Python [PYGL, 2016].

- TCP/IP e conexão segura durante o jogo: durante o jogo, quando há muitas entidades se movendo, há muitas mensagens sendo trocadas na rede. Isto faz que tenhamos lentidão, já que quando utilizamos TCP temos que esperar acabar de receber todos os pacotes na sequência para reconstruir as mensagens enviadas. Quando estamos utilizando conexão segura, temos que enviar muitas mensagens e adicionar informações para garantir a segurança da mensagem. Uma forma de evitar os problemas em comunicação é a mudança de TCP para UDP, utilizando um mecanismo interno para a conferência das mensagens. Em relação a conexão segura, podemos corrigir este problema sem perder a segurança fazendo o uso da conexão segura apenas para o login, gerando uma chave única para conectar ao jogo e para cada mensagem trocada deve ser utilizado um número sequencial codificado com um código de verificação.

- Excesso de mensagens trocadas: para cada mudança de estado das entidades próximas é enviada uma mensagem para todos os jogadores. Se há múltiplas entidades se movendo o tempo todo, isto pode causar muita lentidão, já que há muitas mensagens sendo enviadas. Uma forma de evitar este problema é para cada ciclo de mensagens da *microthread* do jogador fazermos o processamento das mensagens e

o envio em lotes de todas as unidades que tiveram seu estado alterado.

Referências

[CCP] CCP Games. *Stackless Python in EVE* (2007). Disponível em: http://www.slideshare.net/Arbow/stackless-python-in-eve

[CET] Centro Regional de Estudos para o Desenvolvimento da Sociedade da Informação. *Índicadores A4 – Proporção de Domicílios com Acesso a Internet* (2015). Disponível em: http://cetic.br/tics/usuarios/2015/total-brasil/A4/

[GIT] GitHub: Victor R. Cardoso. *AlphaMMO* (2016). https://github.com/labrax/AlphaMMO

[GOF] E. Gamma, J. Vlissides, R. Johnson and R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (1994).

[GRE] J. Gregory. Game Engine Architecture. A K Peters/CRC Press (2009).

[GSA] M. Walker. *Know what to build: Multiplayer Game Types*. Engines of Delight. (2015) Disponível em: https://gameserverarchitecture.com/2015/09/know-what-to-build-multiplayer-game-types/

[GSA2] M. Walker. *Game Server Architecture Patterns*. Engines of Delight. Disponível em: https://gameserverarchitecture.com/game-server-architecture-patterns/

[NYS] R. Nystrom. *Game Programming Patterns. Genever Benning* (2014). Disponível em: http://gameprogrammingpatterns.com/

[PYG] PyGame Team. *PyGame* (2016). Disponível em: http://pygame.org/

[PYGL] PyOpenGL Team. *PyOpenGL 3.x* (2016). Disponível em: http://pyopengl.sourceforge.net/

[PYO] Python Package Index. *PyOpenSSL 16.2.0* (2016). Disponível em: https://pypi.python.org/pypi/pyOpenSSL

[PYP] Python 2.7.2 Documentation. *Pickle* (2016). Disponível em: https://docs.python.org/2/library/pickle.html

[RIC] C. Richardson. *Pattern: Monolithic Architecture* (2014). Disponível em: http://microservices.io/patterns/monolithic.html

[SIL] A. Silberschatz, P. B. Galvin & G. Gagne. Operating System Concepts. Wiley 9th (2013).

[STA] Stackless Python Team. *Stackless Python* (2016). Disponível em:

https://bitbucket.org/stackless-dev/stackless/wiki/Home

[WCP] Woodside Capital Partners. *Video Game Market Report* (2015). Disponível em: http://www.woodsidecap.com/wp-content/uploads/2015/12/WCP-Video-Game-Report-20151104.pdf

[WIK] Wikipedia. *Jogo* (2016). Disponível em: https://pt.wikipedia.org/wiki/Jogo

[WIK2] Wikipedia. *List of videogame genres* (2016). Disponível em: https://en.wikipedia.org/wiki/List_of_video_game_genres

[WIK3] Wikipedia. *Model-view-controller* (2016). Disponível em: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller