5
Ferramenta Proposta e Resultados

Este capítulo trata de uma ferramenta que implementa muitas das técnicas explicadas nas seções anteriores.

5.1
Objetivo

O objetivo da ferramenta proposta nesta dissertação é facilitar a implementação da parte distribuída de jogos para jogadores em massa(*massively multiplayer*) de RPG. Segundo [16], o gênero de jogo MMORPG Fantástico detém 84.8% do mercado. A maioria dos jogos multijogador em massa de RPG possuem dinâmica semelhante, isto é, a maneira de jogar é parecida. Logo, uma ferramenta desse tipo é algo útil e interessante para desenvolvedores interessados em entrar nesse mercado.

A ferramenta implementada consiste em uma biblioteca de vínculo dinâmico(.dll) que pode ser usada em um jogo multijogador em massa. É possível customizar várias partes de seu processamento alterando scripts em Lua usados pela biblioteca.

Todo o processamento distribuído no servidor e no cliente será tratado pela ferramenta. A ferramenta tratará da comunicação entre os servidores e os clientes e também da lógica de jogo que possui uma forte relação com a distribuição, por exemplo, o processamento da movimentação dos jogadores é feito na biblioteca pois facilita o uso de técnicas que reduzem o número de mensagens de movimentação trocadas.

A ferramenta proposta vem com subpartes que tratam da criação do servidor e da criação do cliente.

PUC-Rio - Certificação Digital Nº 0410842/CA

5.2 Servidor genérico

A tecnologia implementada permite a criação de uma rede de servidores, que compartilha o processamento do jogo. Um servidor pode tratar de diferentes processamentos dependendo do seu tipo:

- LOGIN: O servidor trata do login dos jogadores no sistema, e do redirecionamento inicial para o servidor responsável baseado nas posições dos jogadores.
- GAMELOGIC: O servidor trata do processamento de um retângulo do mundo.
- DATABASE: O servidor possui um banco de dados e pode armazenar e pesquisar informações. Apenas um servidor DATABASE pode existir na rede.
- MASTERSERVER: O servidor é responsável pela gerência da rede. Apenas um MASTERSERVER pode existir na rede.

Um mesmo servidor pode ser de mais de um tipo. Por exemplo, um servidor pode incorporar todos os 4 tipos e fazer todo o processamento do mundo nele mesmo. Os servidores DATABASE e MASTERSERVER não recebem conexões de clientes e, portanto, não precisam estar conectados à internet. Apenas os servidores de LOGIN e GAMELOGIC que precisam estar conectados à internet.

O servidor de tipo MASTERSERVER é responsável por manter a lista de servidores. Todo servidor, quando é iniciado, tem especificado em seu script o endereço do MASTERSERVER. Cada servidor então tenta, no início, se conectar ao servidor mestre, enviando também a senha dos servidores, o seu nome e o seu tipo. Após enviar esses dados, o servidor recebe sua identificação e a lista de servidores com os seus respectivos tipos. O servidor então passa a se conectar a cada servidor e a se comunicar com cada um de acordo com o seu tipo. Esse processo é mostrado na figura 5.1. A figura 5.2 mostra uma possível organização de uma rede de servidores.

Todo cliente inicialmente se conecta a um servidor de LOGIN e envia uma identificação avisando que é um cliente tentando se conectar. O servidor responde se o cliente pode realizar login ou não. Se for possível realizar o login, o cliente envia o nome do usuário e senha. O servidor de LOGIN confere esses dados com o servidor de DATABASE e, a seguir, redireciona o cliente para o servidor de GAMELOGIC responsável. Esse processo é mostrado na figura 5.3.

PUC-Rio - Certificação Digital Nº 0410842/CA

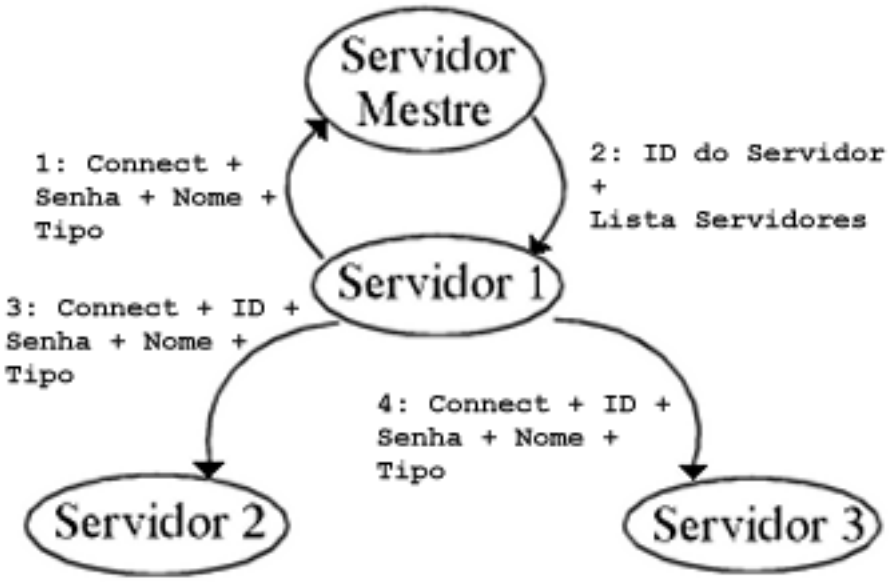

Figura 5.1: Início da comunicação de um servidor com o servidor mestre e com os outros servidores da rede.

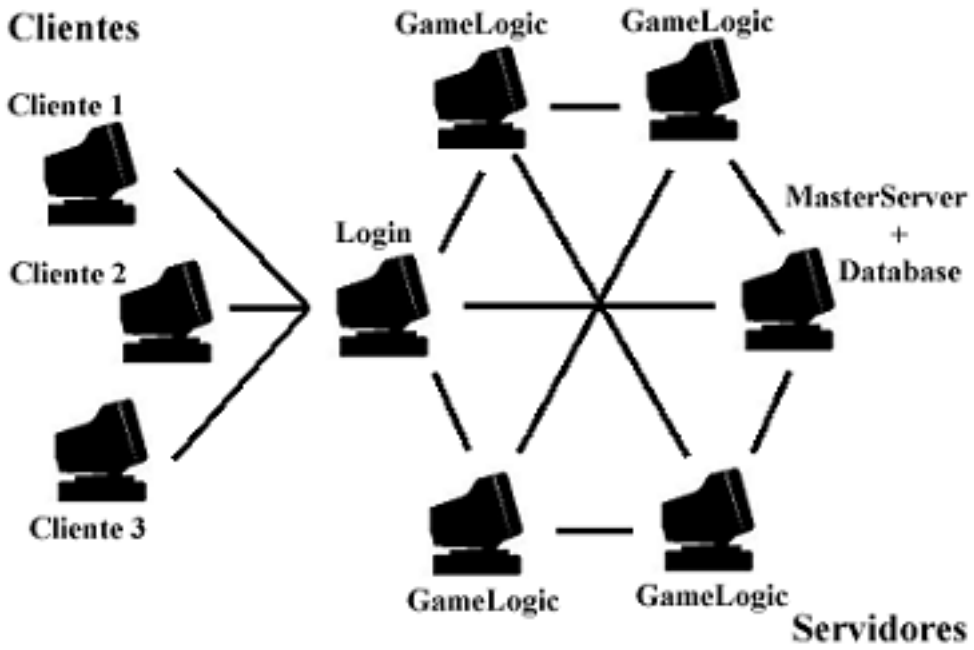

Figura 5.2: A figura mostra uma possível organização para uma rede de servidores.

PUC-Rio - Certificação Digital Nº 0410842/CA

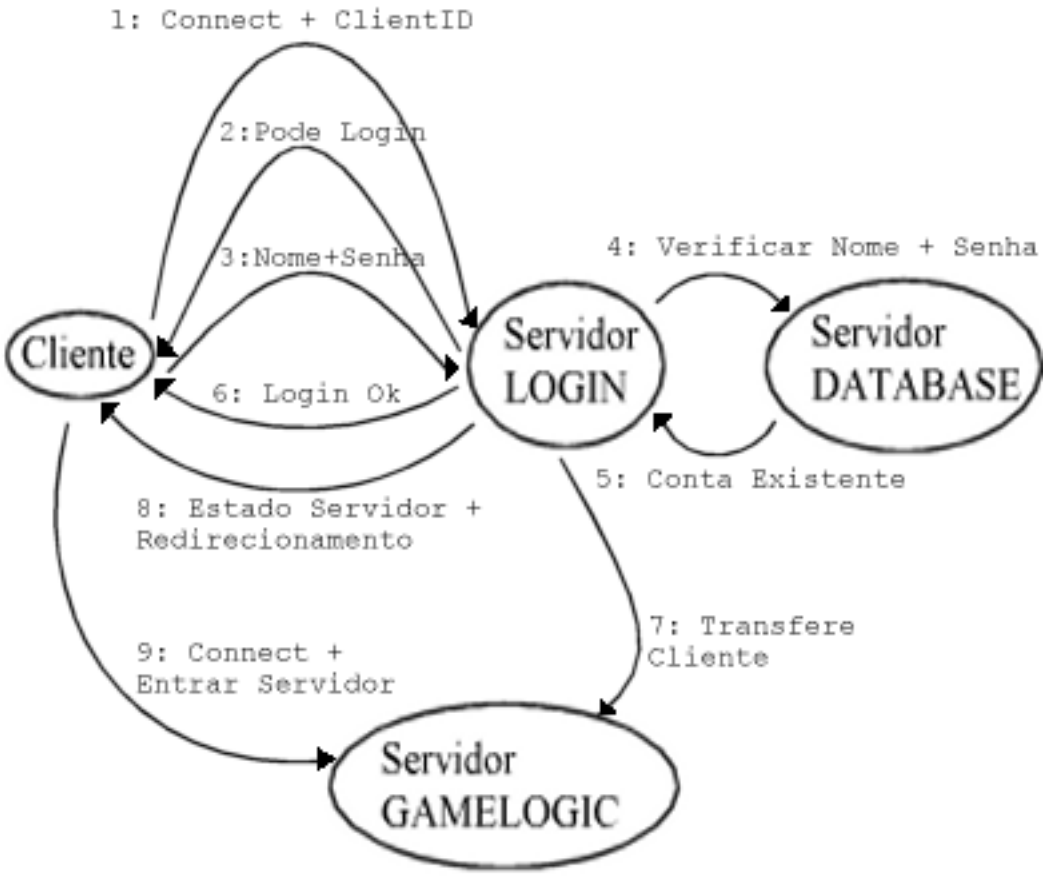

Figura 5.3: O processo de login de um cliente.

Os servidores de GAMELOGIC são responsáveis pelo processamento de toda lógica de jogo que acontece em um retângulo do mundo. Eles se comunicam diretamente com os clientes localizados em sua região e também redirecionam clientes que saíram de seus retângulos.

O servidor de DATABASE serve para atualizar e recuperar informações sobre o estado de jogo do mundo. A ferramenta implementada usa um servidor mySQL instalado no servidor de DATABASE para armazenar os dados do jogo. Todos os outros servidores que precisam acessar a base de dados do jogo interagem diretamente com esse servidor fazendo pesquisas e alterando a base de dados. A aplicação que roda o servidor de DATABASE tem apenas a funcionalidade de indicar para a rede a localização do servidor onde se encontra a base de dados.

A figura 5.4 mostra as classes da ferramenta proposta nesta dissertação. É importante dizer que o *design* mostrado na figura não é o de uma ferramenta completa com todas as partes necessárias para implementar a distribuição de MMORPGs. Diversas partes importantes (como criptografia) não foram incluídas nesse trabalho.

PUC-Rio - Certificação Digital Nº 0410842/CA

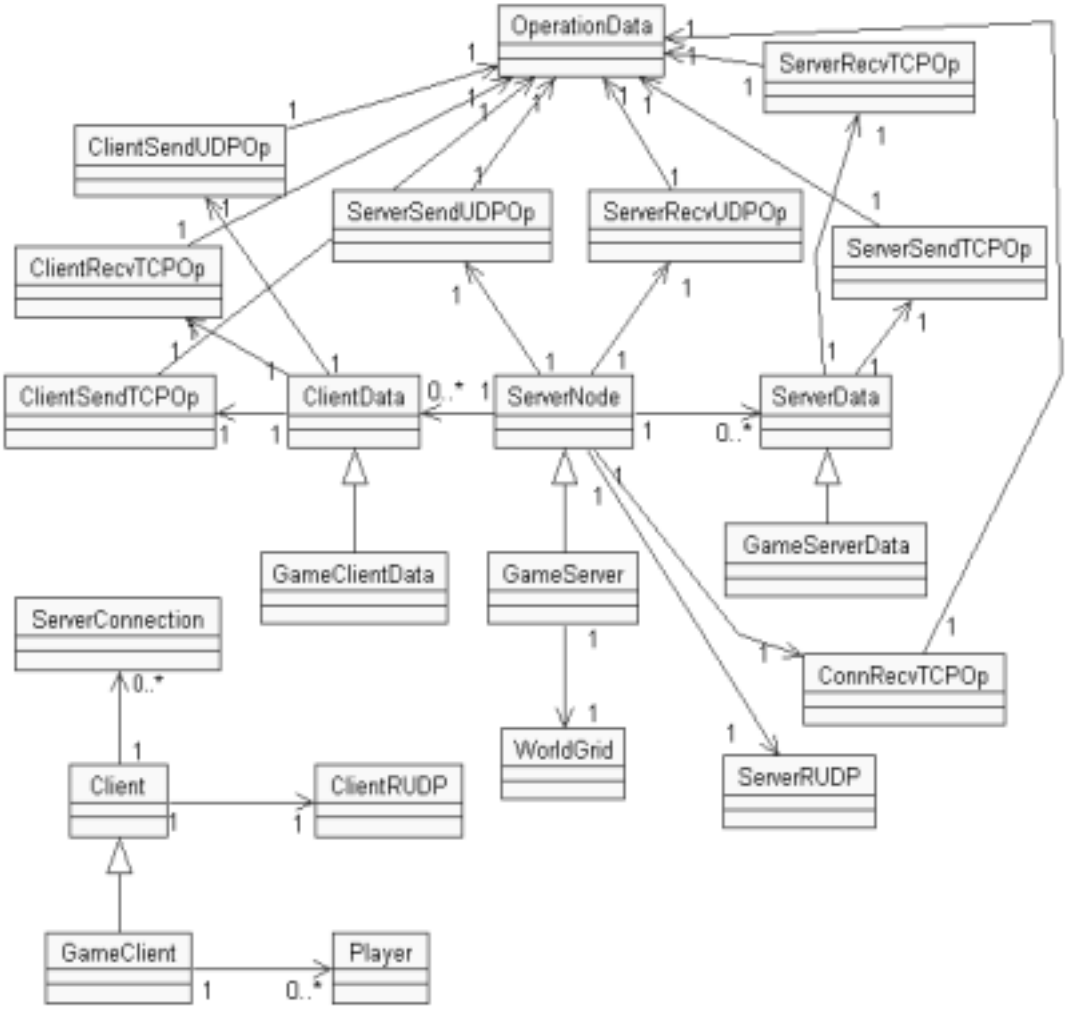

Figura 5.4: A figura mostra as classes da ferramenta implementada.

O servidor é implementado em duas classes principais, a ServerNode e a GameServer. A ServerNode trata da lógica de distribuição mais genérica que quase todo jogo multijogador em massa deve tratar. Isso envolve envio e recebimento de mensagens para cliente e servidores, eventos de clientes se conectando e desconectando, e erros.

É a ServerNode que lida com a API do WinSock e do *IO Completion Ports* diretamente. A tecnologia *IO Completion Ports* é usada pois possui boa escalabilidade como já explicado no capítulo 3. Todos as operações de envio e recebimento de dados usam buffers da aplicação diretamente. Isso foi feito colocando os tamanho dos buffers dos sockets iguais a 0 e realizando operações *overlapped*. O algoritmo de Naggle também é desligado e a própria aplicação agrega mensagens TCP que podem ser enviadas em conjunto para um computador destino.

A classe ServerNode é quem recebe as conexões e as divide em duas listas, a de clientes e a de servidores. Ela se conecta ao *master server* e,

PUC-Rio - Certificação Digital Nº 0410842/CA

após receber a lista de servidores, realiza a conexão com todos eles, criando o *grid* de servidores.

O servidor possui um buffer único para o recebimento de mensagens UDP. Cada cliente e servidor possuem buffers próprios para o envio e recebimento de dados. O uso de múltiplos buffers é recomendado, pois permite que, enquanto uma mensagem é tratada, outros buffers sejam carregados com mensagens que chegam (ou descarregados com mensagens sendo enviadas).

A estrutura dos clientes, com as informações tratadas pelo ServerNode é a seguinte:

```
struct ClientData
{
   SOCKET            SocketTCP;
   SOCKET            SocketSendUDP;
   sockaddr_in       ClientAddr;
   ClientRecvTCPOp   RecvTCP;
   ClientSendTCPOp   SendTCP;
   ClientSendUDPOp   SendUDP;
   ...
   DWORD             SRTT;
   DWORD             RTTVAR;
   int               RetransmitCounter;
   ...
```

SocketTCP armazena um socket TCP conectado do cliente; é usado para receber e enviar mensagens TCP entre o cliente e o servidor. SocketSendUDP é um socket UDP connectado usado para o envio de mensagens UDP para esse cliente. ClientAddr armazena o endereço do cliente. RecvTCP é uma estrutura usada em operações *overlapped* do cliente para recebimento de mensagens TCP. SendTCP tem o mesmo propósito, só que para envio de mensagens TCP. SendUDP também tem esse mesmo propósito, só que para o envio de mensagens UDP. Outros dados de interesse são o SRTT que consiste no tempo de RTT(*round trip time*) estimado, RTTVAR que consiste na variação estimada do RTT entre mensagens e o RetransmitCounter que conta o número de retransmissões de mensagens para esse cliente.

A estrutura dos servidores é semelhante:

```
struct ServerData
```

PUC-Rio - Certificação Digital Nº 0410842/CA

```
{
   int               ServerID;
   int               ServerType;
   sockaddr_in       ServerAddr;
   SOCKET            SocketTCP;
   ServerRecvTCPOp   RecvTCP;
   ServerSendTCPOp   SendTCP;
   ...
```

ServerID é uma identificação do servidor fornecida pelo master server. ServerType especifica o tipo de servidor (Ex: ST_MASTERSERVER). ServerAddr possui o endereço do servidor. SocketTCP possui um socket TCP conectado do servidor. RecvTCP e SendTCP são estruturas usadas em operações overlapped para o servidor.

Todas as estruturas usadas em operações overlapped são mais ou menos semelhantes, a estrutura para operações de recebimento de dados TCP em clientes é mostrada abaixo:

```
struct ClientRecvTCPOp
{
   OperationData  OpData;
   WSABUF         RecvDataBufTCP;
   char           RecvBufferTCP[CLIENT_RECV_BUFFER_TCP_SIZE];
   ClientData   * pClient;
   ...
};
```

OpData é uma estrutura com o seguinte formato:

```
struct OperationData
{
   OVERLAPPED        Overlapped;
   OPERATION_TYPE    OperationType;
};
```

onde Overlapped é a estrutura necessária para realizar operações overlapped. OperationType armazena o tipo de operação sendo efetuado.

Com relação à estrutura utilizada em operações para recebimento de dados TCP pode-se dizer o seguinte: RecvDataBufTCP é um buffer do WinSock que apontará para RecvBufferTCP que, de fato, é o buffer que armazena os bytes recebidos da mensagem TCP; pClient aponta para

PUC-Rio - Certificação Digital Nº 0410842/CA

o cliente responsável por essa operação. Todas as outras estruturas de operações são semelhantes a essa operação, possuindo OpData no começo e um buffer para armazenar bytes a serem transferidos.

Quando deseja-se realizar uma operação overlapped, usam-se as estruturas de operações encontradas em ClientData e ServerData. O exemplo abaixo mostra o envio de uma mensagem TCP para um cliente:

```
ZeroMemory(&pClient->SendTCP.OpData.Overlapped,
         sizeof(pClient->SendTCP.OpData.Overlapped));

r = WSASend(pClient->SocketTCP,
            &pClient->SendTCP.SendDataBufTCP,
            1,
            &SendBytes,
            Flags,
            (OVERLAPPED*)&pClient->SendTCP,
            NULL);
```

Todas as operações *overlapped* usam os buffers que se encontram dentro das estruturas de operação do ClientData e ServerData.

A thread que recebe as notificações do *completion port* também se encontra dentro do ServerNode e possui o seguinte formato:

```
DWORD WINAPI ServerNode::ServerWorkerThread(LPVOID pServer)
{
...
   OperationData * pOp;

   while(true)
   {
      BOOL b;
      b = GetQueuedCompletionStatus(CompletionPort,
                                    &BytesTransferred,
                                    (DWORD*)&Socket,
                                    (OVERLAPPED**)&pOp,
                                    INFINITE);

      ...

      switch(pOp->OperationType)
```

PUC-Rio - Certificação Digital Nº 0410842/CA

```
      {
      case OP_BUFFER_RECV_UDP:
         pServerNode->ProcessBufferRecvUDPEvent(pServerNode,
            CompletionPort,BytesTransferred,Socket,
            (ServerRecvUDPOp*)pOp);
         break;
      case OP_CLIENT_RECV_TCP:
         pServerNode->ProcessClientRecvTCPEvent(pServerNode,
            CompletionPort, BytesTransferred,Socket,
            (ClientRecvTCPOp*)pOp);
         break;
      case OP_CLIENT_SEND_TCP:
         pServerNode->ProcessClientSendTCPEvent(...);
         break;
      case OP_CLIENT_SEND_UDP:
         pServerNode->ProcessClientSendUDPEvent(...);
         break;
      case OP_SERVER_RECV_TCP:
         pServerNode->ProcessServerRecvTCPEvent(...);
         break;
      case OP_SERVER_SEND_TCP:
         pServerNode->ProcessServerSendTCPEvent(...);
         break;
      default:
         pServerNode->OnError("Non identified operation");
      }
   }
...
```

Acima estão todos os tipos de operações possíveis, elas tratam do recebimento de mensagens UDP, recebimento e envio de mensagens TCP de clientes, do envio de mensagens UDP para clientes e do recebimento e envio de mensagens TCP de servidores.

5.2.1 Enviando mensagens UDP de forma garantida

O ServerNode faz uso da classe ServerRUDP que permite que ele envie mensagens UDP com garantia de chegada. O ServerRUDP, no entanto,

PUC-Rio - Certificação Digital Nº 0410842/CA

adiciona funcionalidade à maneira explicada na seção 2.6 de se implementar garantia de chegada.

Existem algumas mensagens em que a garantia de chegada é necessária, porém apenas para as mensagens mais recentes. Um caso já explicado é o da movimentação, como mostrado no exemplo a seguir:

Um servidor recebe uma mensagem de movimentação de um jogador e a envia para um determinado cliente. Pode acontecer que, antes que essa mensagem receba um Ack, uma nova mensagem de movimentação desse mesmo jogador chega. Neste caso, a mensagem antiga de movimentação não importa mais e, portanto, não precisa mais ter garantia de chegada. Só a mensagem mais recente é que deve ter essa garantia. O ServerRUDP implementa justamente essa adição. Ao enviar uma mensagem, é possível se fazer com que todas as mensagens do mesmo tipo para um cliente sejam removidas, ficando apenas aquela última mensagem mais recente. A interface do ServerRUDP que permite isso é a seguinte:

```
void SendReliableUDP(ClientData * pClient,
                     const char * pMsg,
                     short MsgSize,
                     int MsgID = 0,
                     unsigned int PlayerID = 0,
                     bool LatestUpdate = false);
```

onde pClient é um ponteiro para o cliente que se deseja enviar a mensagem com garantia de chegada, pMsg é a mensagem e MsgSize é o tamanho da mensagem. Os próximos parâmetros são usados para remoção de mensagens antigas. MsgID serve para identificar o tipo da mensagem, por exemplo: PT_PLAYER_MOVEMENT é um identificador de mensagens de movimentação no servidor. PlayerID identifica o jogador sobre o qual a mensagem fala a respeito, se existir um. Esse parâmetro existe para permitir que diferentes mensagens de um mesmo tipo, porém de diferentes jogadores, possam ser enviadas para um mesmo cliente. LastestUpdate é uma flag que identifica se é desejado remover as mensagens mais antigas para o pClient com o mesmo MsgID e o mesmo PlayerID. Um exemplo de envio de uma mensagem de movimentação é o seguinte:

```
SendReliableUDP(newClosePlayers[j],
                (const char*)&MovPkt,
                sizeof(MovPkt),
                PT_PLAYER_MOVEMENT,
```

PUC-Rio - Certificação Digital Nº 0410842/CA

```
                MovPkt.PlayerID,
                true);
```

onde newClosePlayers[j] é um jogador que se encontra próximo à algum outro jogador sendo processado. MovPkt é do tipo PlayerMovementPacket e serve para enviar movimentos de jogadores. A informação sendo enviada para o jogador é do tipo PT_PLAYER_MOVEMENT e MovPkt.PlayerID identifica sobre qual jogador a informação pertence.

5.3 Servidor de RPG

O GameServer deriva de ServerNode e é responsável pela implementação da lógica de distribuição específica de um servidor de RPG para jogadores em massa (*massively multiplayer*).

O GameServer trata de login, redirecionamentos dos clientes para os servidores adequados , movimentação, chat e faz acessos à base de dados.

5.3.1 Adquirindo informações de jogadores próximos

O GameServer usa a classe WorldGrid para conseguir rapidamente adquirir informações de objetos próximos a um jogador. Quando um jogador acaba de fazer login, ele recebe os dados de todos os jogadores próximos a ele. Para fazer isso é chamado o método getClosePlayers do WorldGrid. Esse método possui o seguinte protótipo:

```
void getClosePlayers(GameClientData * player,
                     PGameClientData ** closePlayers,
                     int * numOfClosePlayers);
```

onde player é o jogador que se deseja receber os jogadores próximos, closePlayers é o vetor onde são retornados os jogadores próximos e numOfClosePlayers é o número de jogadores no vetor.

O WorldGrid usa o grid explicado nas seções anteriores para armazenar a localização dos jogadores no mundo. O grid é uma matriz 2D no WorldGrid, como mostrado abaixo:

```
class WorldGrid
{
...
```

PUC-Rio - Certificação Digital Nº 0410842/CA

```
private:
   struct GridCell
   {
      list<GameClientData*> playerList;
   };

   GridCell ** worldGrid;
...
};
```

A matriz 2D é composta por elementos do tipo GridCell, que armazenam uma lista de ponteiros para jogadores. A classe GameClientData é uma derivação do ClientData com mais informações (usadas apenas no GameServer). Abaixo são mostrados os dados da GameClientData:

```
class GameClientData : public ClientData
{
...
   unsigned int PlayerID;
   FVector2 Pos;              //position
   FVector2 Dest;             //destination
   float Speed;
   DWORD TimeOfArrival;
   FVector2 Dir;              //direction
...
   int GridX,GridY;
...
};
```

onde PlayerID guarda o identificador do jogador, Pos é a posição, Dest o destino, Speed a velocidade do jogador, TimeOfArrival o momento de chegada e Dir a direção atual do jogador. GridX e GridY armazenam a posição no *grid* do jogador.

A movimentação dos jogadores é implementada com sincronização por tempo de comando de maneira semelhante à explicada na seção 4.2. Logo após efetuar o *login*, o cliente recebe o tempo do servidor e passa a usá-lo para fazer a movimentação dos jogadores.

Durante a movimentação de um jogador, é possível que ele mude de posição no *grid* e com isso encontre novos jogadores que precisam ser alertados de sua presença. A classe WorldGrid fornece o método movePlayer que é usado durante essa movimentação:

PUC-Rio - Certificação Digital Nº 0410842/CA

```
void movePlayer(GameClientData * player,
                PGameClientData ** newClosePlayers,
                int * numOfNewClosePlayers);
```

Esse método, quando detecta que o jogador mudou de posição no grid, pega os novos jogadores próximos ao jogador e os retorna no vetor newClosePlayers, player é o jogador se movimentando e numOfNewClosePlayers é o número de jogadores no vetor.

Uma possível otimização seria usar multicasting, colocando cada região do grid fazendo uso de um grupo de multicasting. Cada jogador seria adicionado aos grupos de multicasting das regiões as quais se encontra próximo. Toda vez que um jogador fizesse um movimento, ao invés do servidor enviar uma mensagem de movimentação para cada jogador próximo, o servidor apenas enviaria uma mensagem para cada região próxima. Todos os jogadores nessas regiões próximas receberiam a mensagem.

A técnica de *grid* foi preferida à técnica de *Quadtree* devido à sua maior simplicidade. Isso, no entanto, traz a necessidade de cuidados ao se definirem os tamanhos dos objetos e das regiões do *grid*, para que não existam aparições repentinas de objetos.

PUC-Rio - Certificação Digital Nº 0410842/CA

5.3.2 Balanceamento

Cada GameServer do tipo GameLogic trata de um retângulo no mundo e redireciona clientes que saem desse retângulo. O balanceamento implementado é semelhante ao balanceamento explicado em seções anteriores. Todo servidor de GameLogic possui até 4 servidores de fronteira. Esses servidores tratam das fronteiras de norte, sul, leste e oeste, e são para eles que são redirecionados os clientes quando passam por alguma dessas fronteiras. Além desses, 4 outros servidores também são acessíveis (nordeste, noroeste, sudeste e sudoeste) quando mais de um fronteira é ultrapassada.

Cada fronteira é expandida para que, antes da fronteira ser cruzada, o cliente já esteja conectado ao servidor de destino.

A função que testa se um jogador cruzou uma fronteira é chamada de TestPlayerAgainstFrontiers, ela possui a seguinte lógica:

```
bool GameServer::
  TestPlayerAgainstFrontiers(GameClientData * pClient)
{
  //Se a PosY do jogador <= MinimoY do Retangulo +
```

```
  //ValorExpandeFronteira
  if((pClient->Pos.y <= (worldGrid.minY + m_FrontierDistTop))
  {
    //Carregar um ClientTransferPacket TransPkt
    TransPkt.PlayerID = pClient->PlayerID;
    TransPkt.Pos = pClient->Pos;
    TransPkt.Dest = pClient->Dest;
    TransPkt.Speed = pClient->Speed;
    TransPkt.TimeOfArrival = pClient->TimeOfArrival;
    TransPkt.Dir = pClient->Dir;

    //Transferir o cliente para o outro servidor
    SendTCP(m_FrontierTopServer,(char*)&TransPkt,
      sizeof(TransPkt));

    //Enviar para o jogador a ordem de conectar ao novo
    //servidor de destino
    ConnectPacket Pkt;
    Pkt.ServerAddr = m_FrontierTopServer->ServerAddr.sin_addr;
    SendTCP(pClient,(char*)&Pkt,sizeof(Pkt));
  }
...
//Para todas as outras fronteiras o código é semelhante
}
```

Quando um cliente recebe a requisição de conexão com o novo servidor, ele requisita a conexão e envia uma mensagem se identificando para o novo servidor. Nesta ocasião, o novo servidor tem as informações do jogador já armazenadas em uma lista, pois elas já tinham sido transferidas de outro servidor.

Quando um jogador se move para além da fronteira do retângulo expandida para fora (dentro do retângulo do servidor de fronteira), ele é removido pelo servidor de sua lista de clientes.

O balanceamento usando retângulos foi selecionado pela sua simplicidade de implementação e eficiência. Porém se muitos jogadores ficarem próximos a fronteiras com 3 servidores, a performance irá degradar numa taxa superior ao caso se hexágonos tivessem sido utilizados.

PUC-Rio - Certificação Digital Nº 0410842/CA

5.4 Movimentação

A movimentação costuma ser o tipo de mensagem que gera mais tráfego num jogo multijogador em massa. Sendo assim, técnicas eficientes para reduzir o número de mensagens de movimentação podem reduzir consideravelmente o número de mensagens enviadas para o servidor e para o cliente.

A movimentação na ferramenta proposta é uma implementação do *dead-reckoning* baseado em objetivos. O objetivo no caso é o local de destino. Quando um jogador deseja se mover para um determinado lugar, ele envia uma mensagem para o servidor requisitando a movimentação. Após fazer isso, mesmo sem resposta do servidor, o jogador já começa a se mover para a posição de destino. A mensagem enviada para o servidor possui a seguinte estrutura:

```
class RequestMovementPacket : public Packet
{
public:
   RequestMovementPacket()
   {
      PacketType = PT_REQUEST_MOVEMENT;
   }

   float DestinationX, DestinationY;
};
```

onde DestinationX e DestinationY são a posição 2D de destino para onde o jogador deseja ir. São apenas 2 posições, porque mesmo em MMORPGs 3D, a movimentação dos jogadores geralmente é em 2D.

São usados floats para definir a posição de destino, mas shorts e até chars poderiam também ter sido utilizados. Uma maneira de se calcular a posição de destino, usando um desses tipos de menor quantidade de bytes, é adicionando a posição enviada na mensagem com a posição mínima em x e y do retângulo do servidor, como mostra a figura 5.5.

Na mensagem de movimentação transmitida do cliente para o servidor, não é enviado um identificador do jogador. Para fazer a identificação do cliente que enviou uma mensagem UDP, o servidor usa o endereço de envio da mensagem. Através do endereço, o servidor chega ao cliente.

O servidor, ao receber a mensagem, começa a mover o jogador e envia para todos os jogadores próximos a ele, incluindo o próprio jogador que

PUC-Rio - Certificação Digital Nº 0410842/CA

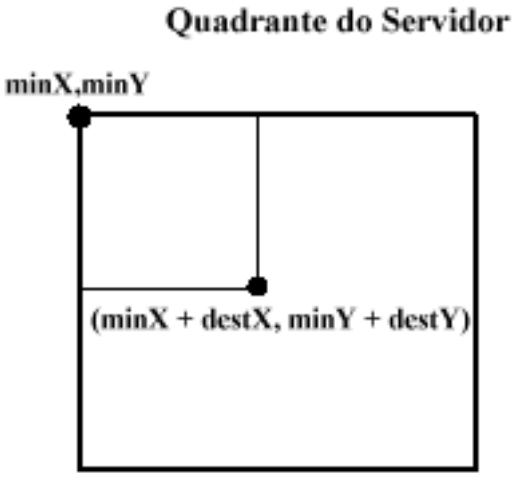

Figura 5.5: Como calcular uma posição de destino usando shorts ou chars.

enviou a mensagem, a mensagem de movimentação e o momento de chegada. Essa mensagem possui a seguinte estrutura:

```
class PlayerMovementPacket : public Packet
{
public:

   PlayerMovementPacket()
   {
      PacketType = PT_PLAYER_MOVEMENT;
   }
   unsigned int PlayerID;
   float DestX,DestY;
   DWORD TimeOfArrival;
};
```

onde PlayerID identifica o jogador que fez a movimentação, DestX e DestY é o destino do jogador, TimeOfArrival é o momento em que o jogador deve chegar ao seu destino. Quando os jogadores recebem essa mensagem, cada um calcula a velocidade necessária para que o jogador chegue na posição no momento certo.

O uso de *dead-reckoning* baseado a objetivos permite que apenas com um x e y seja feita toda a movimentação de um jogador. Usando um outro tipo de *dead-reckoning*, qualquer alteração na direção do jogador teria que ser enviada na rede, o que geraria a necessidade de um número maior de mensagens para realizar uma movimentação.

PUC-Rio - Certificação Digital Nº 0410842/CA

5.5 Cliente

A parte do cliente é implementada por duas classes principais, o Client e o GameClient. O Client, assim como o ServerNode, é responsável por tratar da lógica de distribuição que qualquer cliente de jogo para jogadores em massa (*massively multiplayer*) tem que lidar. Essa classe trata do recebimento e envio de mensagens quaisquer e da conexão com um ou mais servidores.

O método de processamento de mensagens no cliente é bem mais simples que o usado no servidor, principalmente porque os requerimentos de distribuição dos clientes são muito diferentes. O cliente só envia mensagens para o servidor e trata das mensagens recebidas dele. Para esse processamento são usados sockets bloqueantes, com uma thread para recebimento de mensagens TCP e uma thread para recebimento de mensagens UDP.

Ao requisitar um pedido de conexão à classe cliente, uma thread é criada para tratar da conexão. O sucesso ou fracasso da função é retornado na função virtual pura abaixo:

```
virtual void OnConnectionComplete(bool Sucessfull) = 0;
```

A classe GameClient, assim como o GameServer, serve para tratar da lógica de distribuição específica de jogos de RPG para jogadores em massa (MMORPGs). Essa classe não apenas envia mensagens de Login, Chat e movimentação como também calcula a movimentação dos jogadores.

5.5.1 Sincronização por tempo de comando

Ao receber uma mensagem do tipo PlayerMovementPacket (mostrada na seção 5.4), o cliente sempre calcula a velocidade do jogador baseado no tempo de chegada especificado pelo servidor e enviado na mensagem. Abaixo é mostrada o trecho que calcula essa velocidade

```
(*it).Speed = d / (float)(p->TimeOfArrival - m_ServerTime);
```

onde d é a distância que deve ser percorrida para chegar ao destino, p->TimeOfArrival é o tempo de chegada especificado na mensagem recebida e m_ServerTime armazena o tempo do servidor localmente.

Com isso, mesmo que a mensagem de um jogador demore para chegar a um cliente, se esse cliente tiver o tempo do servidor (m_ServerTime) próximo

PUC-Rio - Certificação Digital Nº 0410842/CA

do tempo do servidor dos outros clientes, o jogador vai adquirir a velocidade necessária para que ele chegue no mesmo momento que em todos os outros clientes.

5.6 Resultados

A ferramenta desenvolvida nesta dissertação faz uso de técnicas e tecnologias que permitem gerenciar milhares de jogadores conectados em um mesmo jogo. Na figura 5.6 é mostrado um exemplo de uso da ferramenta implementada. A imagem de cima é uma aplicação que roda um servidor e renderiza um mapa que mostra a posição dos jogadores no servidor. A aplicação de janela interage com a classe GameServer para acessar as posições dos jogadores. Na imagem de baixo é mostrado um cliente que faz uso da classe GameClient para poder movimentar um jogador.

É dificil comparar a ferramenta desenvolvida nesta dissertação com outras já implementadas. De todas as ferramentas encontradas relacionadas ao desenvolvimento de MMORPGs, ou elas possuíam o mesmo propósito da ferramenta desenvolvida mas não eram gratuitas ou elas eram gratuitas mas não possuíam o mesmo propósito da ferramenta desenvolvida. Duas ferramentas gratuitas que podem ser usadas na criação de jogos multijogador em massa são o ICE [6] e o WorldForge [32].

No exemplo da figura 5.6, o servidor mostra o posicionamento de dois jogadores em seu mapa e muda esse posicionamento de acordo com a movimentação dos jogadores. O cliente renderiza um avatar do jogador em 3D e todos os avatares dos jogadores próximos. A única ação possível de ser realizada no exemplo é a movimentação. A linguagem utilizada para a implementação do exemplo é C++. As máquinas usadas para executar o exemplo são: Pentium IV 2.26 GHz, 256 megabytes de RAM, 18.9 Gigabytes de disco rígido e Celeron 1.2 GHz, 128 megabytes de RAM, 18.6 Gigabytes de disco rígido.

PUC-Rio - Certificação Digital Nº 0410842/CA

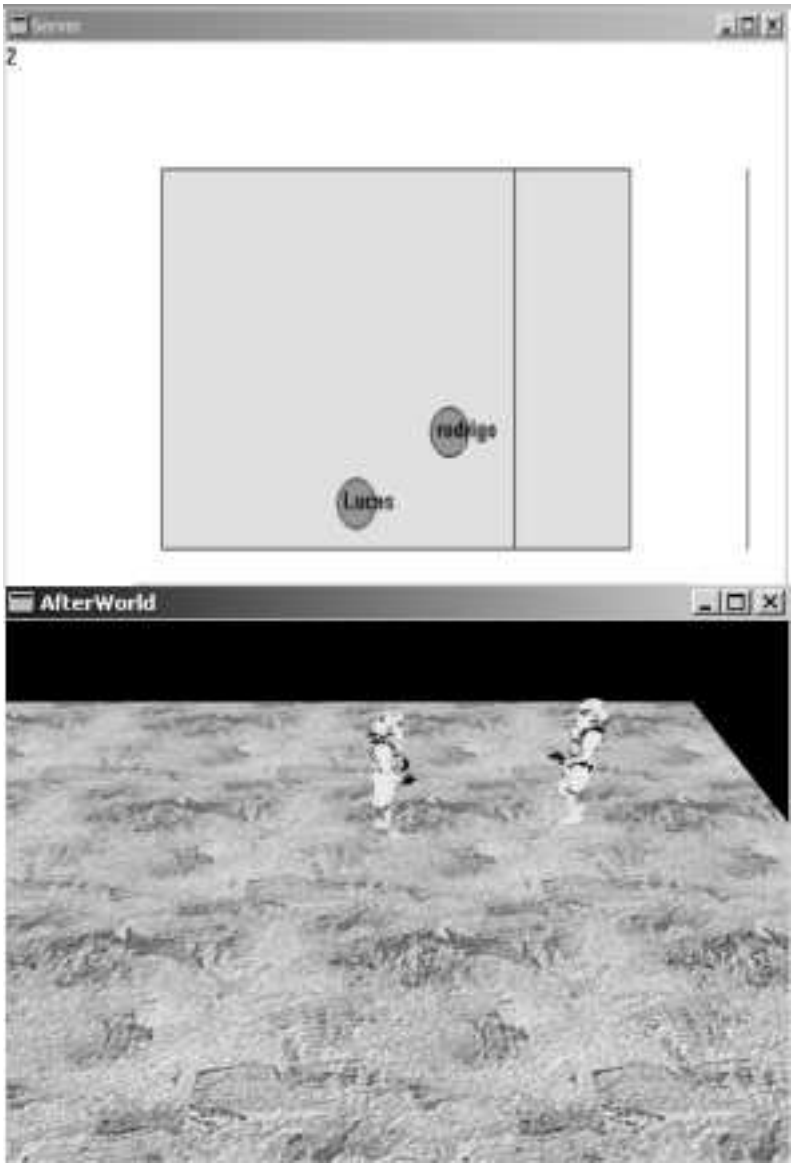

Figura 5.6: A figura mostra um servidor e um cliente que fazem uso da ferramenta proposta nesta dissertação.

PUC-Rio - Certificação Digital Nº 0410842/CA