Desenvolvimento de um protocolo de rede local P2P e de um plugin para o motor de jogos Unity 3D

Trabalho de conclusão de curso para obtenção do grau de Bacharel em Ciência da Computação pelo IME-USP

Renato Scaroni

Orientação: Daniel Macedo Batista

São Paulo, Setembro de 2014

Conteúdo

1 Introdução **3**
- 1.1 Motivações . . . 3
- 1.2 Objetivos . . . 4

2 Considerações técnicas **5**
- 2.1 Conceitos básicos de redes e aplicações para redes . . . 5
  - 2.1.1 O protocolo IP . . . 6
  - 2.1.2 Os protocolos UDP e TCP . . . 7
  - 2.1.3 Sockets . . . 8
- 2.2 Conceitos de programação para jogos e Unity 3D . . . 9
  - 2.2.1 Noções de programação para jogos . . . 10
  - 2.2.2 A Unity 3D . . . 11

3 Protocolo proposto **14**
- 3.1 Ideia geral do protocolo . . . 14
- 3.2 Detalhes do protocolo . . . 15
  - 3.2.1 Organização . . . 15
  - 3.2.2 Transições de estado . . . 15
  - 3.2.3 A terceira fase do protocolo . . . 16

4 Implementação **17**
- 4.1 Base da implementação . . . 17
- 4.2 Organização do código . . . 17
- 4.3 Alguns detalhes importantes . . . 19
- 4.4 Desafios . . . 20
- 4.5 API e integração com projetos da Unity 3D . . . 21

5 Conclusão e considerações finais **22**

1 Introdução

No contexto de comunicação de rede para jogos eletrônicos *multiplayer*, o modelo de aplicação mais utilizado é o de cliente servidor. Nesse modelo sempre existe um papel bem definido para cada máquina na rede, sendo uma delas um servidor e as demais clientes. Por mais que possam existir configurações em que existem mais de um servidor, esse modelo sempre vai existir uma clara distinção bem definidas entre clientes e servidores. Nesse contexto, o servidor dedica-se, em geral, a gerenciar a troca de informação entre os jogadores e guardar informações comuns a todos como dados sobre o cenário de jogo e outras informações gerais. Essa abordagem possui diversos pontos positivos como maior segurança em relação às mensagens enviadas e facilidade na descoberta de novos jogadores, mas apresenta também vários pontos negativos, em especial o fato da comunicação ser centralizada na figura do servidor, o que pode ser perigoso pois qualquer falha no servidor pode implicar em falhas na seção de jogo.

Outro modelo de aplicação para comunicação em jogos é o modelo *Peer-to-Peer* ou P2P. Neste modelo, não há as figuras explícitas de cliente e de servidor. Todas as máquinas que participam da comunicação podem se comportar como clientes ou servidores em diferentes instantes de tempo. Nesse caso, mensagens que precisam ser trocadas entre os participantes são enviadas diretamente. Um ponto positivo dessa abordagem é a o fato de não haver um ponto único de falha, porém há diversos desafios envolvidos como por exemplo a necessidade de manter consistência entre todos os participantes sobre o estado atual do jogo e sobre os jogadores ativos no momento. Este trabalho se dedicará a desenvolver um protocolo de comunicação para uma rede distribuída P2P, ou seja, no qual todo jogador comunique-se diretamente com os demais e possa ser capaz de descobrir de forma autonoma novos jogadores em uma rede local.

1.1 Motivações

No contexto de jogos, vem crescendo muito atualmente o segmento de jogos casuais. Esses jogos baseiam-se geralmente em mecânicas simples, com partidas rápidas e objetivos simples e bem claros como derrubar porcos verdes com passarinhos lançados com estilingues ou cortar frutas que caem de baixo para cima na tela antes que atinjam o final da mesma. Porém ainda existem poucos jogos deste tipo com suporte a multiplayer, mesmo que em dispositivos não portáteis.

Um dos problemas quem dificultam que possam existir jogos com deste tipo de jogo é a falta de opção em termos de soluções de rede que independam da criação de servidores para existir, uma vez que a criação de um servidor para um jogo de partida rápida pode representar um incoveniente para os jogadores.

Como tentativa de atacar esse problema, este trabalho foca num contexto em que os aparelhos onde serão jogados os jogos estejam conectados em uma mesma rede local. Essa recorte permite que possa ser desenvolvida e implementada uma solução no tempo designado para este trabalho, uma vez que podem existir uma série de variações desse problema dependendo da rede em que se está conectado, do aparelho que se usa para jogar, entre outros.

Por fim, a escolha da Unity como plataforma de desevolvimento deste plugin se deve a dois fatos: O primeiro é o fato da Unity ser uma das *game engines* (mais detalhes sobre *game engines* na seção 2.2.1) mais utilizadas por desenvolvedores independentes no mundo. O segundo fato refere-se ao fato do módulo de redes da Unity ser muito precário e até um tanto instável, além de não permitir configurações de rede em que existam mais de um servidor, o que impede a criação de uma rede P2P.

1.2 Objetivos

Tendo em vista o que foi dito na seção anterior, este trabalho tem 2 objetivos principais:

1. Desenvovimento de um protocolo de rede simples capaz de fazer cada nó da rede descobrir outros novos de forma independente, bem como garantir ao máximo sincronia entre os dados trocados entre os peers durante o jogo.
2. Desenvolvimento de um plugin que implemente o protocolo desenvolvido acima para ser usado com o motor de jogos *Unity 3D*.

2 Considerações técnicas

2.1 Conceitos básicos de redes e aplicações para redes

Antes de relatar o trabalho realizado durante a elaboração do protocolo e sua implementação em forma de plugin, é importante mostrar um pouco sobre alguns conceitos importantes ligados às bases de comunicação e aplicações para redes.

Antes de tudo é importante lembrar que a comunicação de redes costuma ser dividida em camadas, sendo cada uma responsável por um aspecto diferente da comunicação. Segundo o padrão *OSI* (*Open System Interconnection*[2]), existem 7 camadas sendo elas:

1. **Física** - Consiste nos elementos que conectam fisicamente a rede, como cabos, ondas de rádio, fibra ótica, etc.
2. **Enlace** - Responsável por codificar e decodificar os dados que circulam pela rede em bits.
3. **Rede** - Responsável por transmitir os pacotes de um nó a outro da rede. Nesta camada são feitas as rotas de trasmissão de um ponto a outro em uma rede. Aqui que se aplica o protocolo IP(*Internet Protocol*), do qual falaremos um pouco adiante.
4. **Transporte** - Esta camada dedica-se a controlar o fluxo de envio de dados, ou seja, como os nós da rede devem enviar seus dados, se deve houver algum tipo de checagem quanto a um nó ter ou não recebido o pacote .
5. **Sessão** - Responsável por estabelecer e gerenciar conexões entre aplicações.
6. **Apresentação** - Onde se converte os dados que passarão pela camada de transporte. Como máquinas diferentes podem utilizar codificações, é necessária uma normalização dos dados antes que possam trafegar pela rede.
7. **Aplicação** - Responsável por definir os serviços que serão implementados pela aplicação, ou seja, processar os dados recebidos dos demais nós da rede e o que enviar de volta.

O protocolo desenvolvido neste trabalho trabalha especificamente com as camadas 5, 6 e 7 desta divisão, pois define quando e o que deve ser enviado para cada nó da rede, bem como definir uma forma de criar conexões entre estes e converter os dados recebidos para strings em padrão ascii.

2.1.1 O protocolo IP

Este protocolo é o responsável por definir a identificação de máquinas conectadas a uma rede e algumas convenções referentes a estrutura dos pacotes a serem enviados, bem como alguns detalhes de como as máquinas na rede devem lidar com eles. Existem atualmente duas versões do protocolo em uso: a versão 6, conhecida com IPv6, e a versão 4, conhecida como IPv4. Porém, devido ao fato da versão 6 ser ainda muito pouco utilizada, em especial num contexto de redes locais, a versão a ser tratada neste trabalho será sempre a IPv4. Informações adicionais sobre especificações dessas duas versões do protocolo podem ser encontradas na referência [1] e nos RFCs: 791 [3], referente ao IPv4, e 2460 [4] e 4291 [5], referentes ao IPv6.

O protocolo IP organiza os pacotes de dados a serem enviados em *datagramas*, que consistem em arquivos contendo um cabeçalho e os dados em si a serem enviados ao destinatário. O cabeçalho possui diversas informações como origem e destino da mensagem, o protocolo de transporte utilizado, a versão do protocolo IP utilizado, um identificador da mensagem, entre outros.

Porém a parte que mais interessa a este trabalho é a parte de endereçamento. O IPv4 atribui a cada interface conectada a rede um endereço. Uma interface de rede é qualquer dispositivo que conecte uma máquina à rede, no caso deste trabalho podem ser placas de rede, tanto cabeadas quanto *wifi*, conexões a roteadores e swithces e pontos de acesso sem fio. O endereço atribuído a uma interface é um número de 32 bits composto por 4 octetos, isto é, quatro números de 8 bits variando de 0 a 255. Em geral costuma-se representar um endereço IP como quatro números separados por pontos, por exemplo 192.168.1.1. Alguns endereços são reservados, para este trabalho o mais importante deles é o com final 255, que está associado a *broadcast*, isto é, quando um roteador recebe uma mensagem nesse ip ele redistribiu essa mensagem para todos os Hosts da rede.

Em termos técnicos constuma-se chamar genericamente, qualquer interface que se conecte a rede de um Host. O conjunto de um roteador conectando um conjunto de hosts é chamado de sub-rede [6]. Porém, segundo o que foi dito no parágrafo anterior, interface de conexão a roteadores também se conectam na rede, que por sua vez podem estar conectados a outros hosts formando outra sub-rede. Nesse caso, é necessário uma forma de descobrir se cada host de uma rede é um host mesmo ou uma sub-rede.

Por isso cada sub-rede tem associada a ela uma máscara de sub-rede, que consiste em um número que indicará como interpretar um endereço de IP. Esse número também será composto por 4 octetos e indicará que faixas de endereço referen-se a hosts mesmo e que faixa de endereço se referem a sub-redes. Por padrão, em geral, lida-se com 3 classes de sub-redes, que em geral cobrem boa parte dos casos de redes locais, sendo essas:

- Classe A: 255.0.0.0 (primeiro octeto indica sub-rede e os demais o host)
- Classe B: 255.255.0.0 (dois primeiros octetos indicam sub-rede e os demais o host)
- Classe B: 255.255.255.0 (três primeiros octetos indicam sub-rede e o último indica host)

Porém existem mascaras de sub-rede mais complexas para o caso de redes com muitas sub-redes como a internet. Maior detalhes podem ser encontrados no capítulo referente a sub-redes de [7]

2.1.2 Os protocolos UDP e TCP

Estes protocolos atuam na camada de transporte, isto é, são responsáveis por denifir como a troca de mensagens será feita entre dois processos através de uma rede. Note que a identificação de um host na rede, ou o caminho que um pacote fará para chegar de um host a outro, são detalhes definidos pelos protocolos de camada de rede. Os protocolos da camada de transporte tem como função fazer com que uma mensagem gerada por um processo executando em um host A chegue a um processo executando no host B.

Para que essa comunicação aconteça, cada programa que utiliza comunicação de rede em uma máquina precisa estar associado a um protocolo e uma porta, que consiste em um número que identifica os programas conectados a rede usando um determinado protocolo. Suponha que existam dois programas executando em uma máquina, se tivermos um que utiliza protocolo UDP e está conectado na porta 8080 ele será identificado como diferente de outro que utiliza TCP na mesma porta. Porém se tivermos dois programas que utilizem um mesmo protocolo, seus números de porta tem que ser obrigatoriamente diferentes.

O protocolo UDP é o mais simples dos protocolos de camada de transporte. O cabeçalho de seus segmentos possuem apenas 8 bytes e quatro campos: porta do host de origem, porta do host de destino, tamanho do pacote e um checksum do segmento. Ele não possui nenhum tipo de serviço adicional associado, isto é, um pacote enviado utilizando este protocolo será apenas enviado de um processo a outro, não é feito nenhum tipo de checagem para garantir o recebimento da mensagem ou a integridade da mesma além do checksum no cabeçalho.

Já o protocolo TCP executa de uma forma muito mais complexa, pois possui diversos outros serviços associados. Em particular, ele garante que haverá uma troca confiável de mensagem entre dois processos, através do uso de números de sequência nas mensagens, indicadores de recebimento (identificado em seu cabeçalho por um campo de 32 bit chamado *acknowledgment number* e um outros de um bit) e contadores de tempo. Por isso o cabeçalho de um segmento TCP é muito maior do que o de um UDP, tendo de 20 a 24 bytes.

Devido a todo esse cuidado com o envio da mensagem, um segmento TCP é muito maior que um UDP, o que faz com que só por isso já seja muito mais lento trocar mensagens por TCP do que por UDP. Além disso, o protocolo de checagem de mensagens do TCP o torna ainda mais lento em relação ao UDP embora muito mais confiável. Dessa forma, um parâmetro comum para a escolha de um protocolo a ser utilizado na camada de transporte é o tempo do conexão.

Por exemplo, em geral aplicações que precisam de resposta rápida, como streaming de mídias em geral, mensagens instantâneas e jogos normalmente fazem sua comunicação baseada em UDP e não em TCP, enquanto outros como navegadores de internet, e-mail e clientes FTP, uma vez que a integridade da informação é mais importante do que o tempo de transmissão da mesma.

2.1.3 Sockets

Voltando ao problema de comunicar procesos através da rede, é importante que haja garantia de que o programa poderá ler as informações recebidas por um protocolo em uma determinada porta. Essa leitura é feita através uma entidade chamada socket.

O conceito exato de sockets varia um pouco dentro da literatura, mas a parte importante é que essa entidade funciona como uma interface de comunicação de rede através do qual um programa poderá ler ou enviar informações que circularão pela rede. Um socket está sempre associado a um IP, um número de porta e um protocolo de camada de transporte.

Do ponto de vista de código, operações com sockets funcionam de forma semelhante a operações com arquivos. No caso específico do C# na plataforma .NET, pode-se trabalhar diretamente com sockets ou utilizar as classe UdpClient e TcpClient que cuidarão de encapsular somente a parte de criação de sockets. Abaixo temos um exemplo de como criar um socket usando a classe socket para escutar uma mensagem e reenviá-la ao outro host:

```
//Inicializa as informacoees do host
IPAddress ipAddress = hostIp;
IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);

// Cria socket TCP/IP.
Socket listener = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.Tcp );

//Liga o socket e escuta por conexao
listener.Bind(localEndPoint);
listener.Listen(10);

//cria um socket de escrita
Socket handler = listener.Accept();

//Recebe dados.
string data = null;

//Guarda dados em buffer e decodifica-os
byte[] bytes = new Byte[1024];
while (true)
{
    bytes = new byte[1024];
    int bytesRec = handler.Receive(bytes);
    data += Encoding.ASCII.GetString(bytes,0,bytesRec);
    if (data.IndexOf("<EOF>") > -1)
    {
        break;
    }
}

// Reenvia para o cliente.
byte[] msg = Encoding.ASCII.GetBytes(data);

handler.Send(msg);
handler.Shutdown(SocketShutdown.Both);
handler.Close();
```

2.2 Conceitos de programação para jogos e Unity 3D

Apesar de os principais objetivos deste trabalho estarem ligados ao desenvolvimento do protocolo e sua implementação, foi necessário um estudo acerca da plataforma em

que o plugin será desenvolvido. É importante ressaltar que várias decisões a respeito da implementação foram tomadas pensando em fazer um bom uso de recursos ligados a conceitos de programação para jogos e a recursos específicos do motor de jogos utilizado, a Unity 3D.

2.2.1 Noções de programação para jogos

Todo jogo consiste no fundo de uma aplicação interativa, onde cada interação do usuário deverá ser interpretada pelo jogo e poderá gerar alterações na saída que o jogo mostra na tela, por isso a estrutura básica de qualquer jogo é um ciclo. Esse ciclo, chamado *Game loop* consiste de três etapas básicas: inicialização, atualização e desenho. Em geral costuma-se referir a estas etapas por seus nomes em inglês, ou seja, *Start*, *Update* e *Draw* ou *Render*.

A primeira fase é realizada apenas uma única vez por entidade que compõe o jogo, em geral quando o jogo inicia. Por exemplo, é definido o número inicial de vidas de um personagem, posição inicial dos inimigos, localização de itens, etc.

Após a primeira fase ser executada, as demais ocorrem repetidas vezes de forma consecutiva, sempre seguindo o ciclo *Update* $\rightarrow$ *Draw* $\rightarrow$ *Update* $\rightarrow$ *Draw* $\rightarrow$ ... . Note que se por algum motivo algum motivo algum elemento novo aparece no jogo (por exemplo se aparece *boss*, ou algum inimigo especial que não havia sido inicializado) ele obedecerá ao *Game loop*, ou seja, será inicializado após um draw e depois seguirá o ciclo como os demais elementos do jogo já em cena.

Na fase de Update é onde será tratada toda interação do usário e recalculado qualquer comportamento de agentes autônomos no jogo (Inteligência artificial). Cálculos de física e qualquer outra simulação baseada em tempo ou que necessite de interação de usuário, bem mensagens recebidas por comunicação de rede ou qualquer outro tipo de informação necessária para geração de imagens que devem ser mostradas para o jogador serão processados nessa etapa.

Note que é de extrema importância que esses dados sejam obtidos de forma mais eficiente possível para não atrasar a ocorrência da terceira fase. Também é importante que essa fase dure sempre mais ou menos o mesmo tempo. Um atraso nessa fase pode significar que menos imagens serão geradas por segundo no jogo, o que prejudica muito a experiência do jogador.

A medida da taxa de imagens geradas por segundo (*Frames per second* ou *FPS*) é uma medida extremamente importante em um jogo, uma vez que uma lentidão quebra o ritmo do jogo. Para visulizar bem a importância desta medida podemos imaginar um jogo de corrida, por exemplo. A sensação de velocidade é impossível de ser mantida se poucas imagens são geradas por segundo. Para gerar uma sensação de movimento suave, o olho humano precisa de pelomenos 25 FPS. É muito importante que acima de qualquer coisa, que o FPS mantenha-se o mais constante possível, quedas bruscas de FPS podem causar uma impressão muito ruim para o jogador.

No caso de comunicação de rede há ainda outro grande problema relacionado a jogos que é garantir que as mensagens cheguem rápido o suficiente e na ordem cronológica correta para que a fase de Draw possa gerar uma imagem que seja coerente para todos os jogadores. Caso isso não aconteça pode acontecer uma situação em que um jogador jogando um jogo de tiro vê um inimigo em sua frente, quando ele já não está "de fato"mais lá e não consegue acertá-lo.

Outro conceito essencial relacionado a programação de jogos que será extensivamente utilizado daqui para frente nesse trabalho é o de *game engine*, ou motor de jogos, se traduzido para o português. Uma *game engine* é uma plataforma que cuidará de gerenciar todos os aspectos relacionados a hardware (tratar requisições de teclado, controles em geral, requisições à placa de vídeo, interface com placas de rede, placa de som, etc), além de organizar e gerenciar a execução do *game loop*, para que o programador possar focar em desenvolver a parte lógica do jogo sem precisar se preocupar com essas questões mais gerais e de baixo nível.

Uma coisa muito importante sobre *game engines* é que elas, em geral, para otimizar a execução das partes mais pesadas (como cálculos de iluminação de cena, por exemplo), são escritas em alguma linguagem de mais baixo nível, geralmente em C++. Porém, para facilitar, elas dão suporte ao uso de linguagens de mais alto nível para as partes de lógica do jogo, mecânica de jogo, gerenciamento de elementos em cena, e outras tarefas menos pesadas. Essas linguagens são chamadas linguagens de script, embora as vezes, sejam usadas para jogos linguagens como C#, que apesar de não serem tradicionalmente para esse fim, são interpretadas, de alto nível, mas possuem recursos muito importantes como orientação a objetos.

2.2.2 A Unity 3D

A Unity 3D é uma *game engine*, muito utilizada, em especial desenvolvedores independentes de jogos por possuir um plano de licensa de uso gratuita, e uma comunidade extremamente ativa e participativa. A Unity é toda desenvolvida em C++ e suporta como linguagens de script C#, javascript e Boo, sendo que os próprios desenvolvedores

recomendam e incentivam o uso de C# pelo fato de ser muito mais versátil e poderosa que as demais. Por isso as considerações feitas nesse trabalho, bem como o desenvolvimento do plugin se deu nessa linguagem.

O desenvolvimento de um jogo utilizando a Unity é feito de forma a ser o mais intuitivo e facilitado possível para que os programadores possam focar em desenvolver melhor a lógica e a mecânica de seus jogos se preocupando o mínimo possível com gerenciamento de recursos e fluxo de jogo. Por isso, ela adota uma estrutura orientada a Asset, isto é, sua programação é orienteda aos elementos que compõe a cena de um jogo. Por exemplo, para criar um jogador por exemplo, basta que o programador crie um objeto do tipo *GameObject*, que será a representação de um elemento que deverá aparecer na cena do jogo. Em seguida basta adicionar os componentes de gráfico (materiais, malhas poligonais ou *sprites*), e scripts de comportamento para tratar a interação do usuário e fazê-lo agir da forma desejada.

Para que haja interação de um *GameObject* faça parte do *game loop*, ele deve ter a ele associado pelomenos um script que herde da classe *MonoBehaviour*. Essa classe possui vários métodos que a classe filha pode implementar e que fará com esta seja levada em conta no *game loop*. Em especial deve haver pelomenos uma implementação de um método de Start e de um de Update. A classe *MonoBehaviour* possui mais de um método associado a essas fases do *game loop*, por exemplo Update e LateUpdate. O fato de existir mais de um método para uma mesma fase do *game loop* se deve ao fato da Unity querer dar liberdade ao programador de, dentro do possível, escolher em que momento do game loop deseja fazer suas computações.

Para dar mais liberdade ao programador, a Unity permite que seja usada toda as ferramentas da linguagem que for escolhida como script. No caso do C# algumas ferramentas são realmente úteis como o uso métodos delegate, que são usados pela Unity para construir um sofisticado sistema de eventos. Os eventos da Unity funcionam de forma muito simples: basta que uma classe qualquer em C# implemente um método para lidar com ela e adicione esse método numa lista de métodos a serem chamados. Note que apesar de a classe que criar o evento precisar ser um MonoBehaviour, qualquer classe de C# que pertença ao projeto poderá definir um método Handler para lidar com qualquer evento. Isso facilita a integração e simconização de partes paralelar ou assíncronas do código. No caso foi um recurso muito útil para lidar com mensagens de rede.

Outro ponto interessante de abordar nessa seção é a parte de redes. Na Unity existe um módulo responsável por ligar com gerenciamento de rede em geral, porém este módulo é extremamente limitado, não permitindo por exemplo a existência de mais de um host que faça papel de servidor, o que pode deixar o programador limitado. As deficiências

deste módulo são conhecidas dos desenvolvedores, que estão planejando para as próximas versões uma reformulação total deste módulo.

3 Protocolo proposto

A primeira etapa do trabalho foi realizar uma série de testes para descobrir se a Unity lidaria bem com um módulo de redes que funcionasse indepente do módulo padrão já existente. Uma vez que os testes de viabilidade foram postivos, é hora de pensar em como será o protocolo.

3.1 Ideia geral do protocolo

A partir do objetivo a ser alcançado com o protocolo, que é permitir jogos em uma rede local em um modelo P2P, ou seja, sem a necessidade de um nó agindo como servidor dedicado, tem-se as seguintes ações que devem ser possíveis com o protocolo:

- Quando o programa é iniciado lança-se uma nova thread que ficará escutando uma porta TCP esperando clientes se conectarem. A conexão de um novo nó será informada aos demais através do disparo de uma mensagem de *broadcast*, refeito periodicamente. Essa mensagem será referida daqui para frente como mensagem de *Alive*.
- Quando recebe uma mensagem de *Alive*, o programa tomará a atitude de colocar o peer que a enviou na lista de usuários ativos, caso ele ainda não esteja nesta. A ausência dessas mensagens indica uma falha ou uma saída espontânea do nó e em ambos os casos o nó deve ser removido da lista de nós disponíveis da rede.
- Os jogadores, uma vez que conseguem se enxergar, podem chamar um subconjunto de todos os peers na rede para uma seção de jogo.
- Durante a seção de jogo, toda a comunicação será feita utilizando protocolo UDP, uma vez que este protocolo da camada de transporte tende a permitir uma comunicação mais rápida do que o protocolo TCP, como visto na seção 2.1.2 deste trabalho. A utilização do TCP poderia levar a estados incoerentes nos nós por contra do atraso inerente ao protocolo.
- Durante o jogo deve sempre ser guardadas informações de cada outro jogador conectado e apenas alterá-lo quando receber um pacote pelo socket UDP correspondente. Quais informações serão guardadas será uma escolha do programador dependendo do jogo desenvolvido.

- Garantir que na seção de jogo as informações passadas para o jogador serão coerentes.

3.2 Detalhes do protocolo

3.2.1 Organização

O funcionamento do protocolo será tratado como uma máquina de estados, composta por três estados precedidos por uma fase de inicialização:

- **Inicialização**: Simplesmente inicia os canais de comunicação necessários para recebimento e envio de mensagens em broadcast no modo standBy. Essa fase de inicialização somente será executada uma vez para cada vez, quando aplicativo for iniciado. Por isso não será contada como um estado propriamente dito. Na prática ele será uma parte do estado de Stand by que será executado apenas uma vez.
- **Fase 1 - Stand by**: Nessa fase os jogadores descobrirão novos nós na rede. Essa descoberta se dá através das mensagens de Alive descritas na seção 3.1. Essa fase termina quando algum dos nós da rede envia uma mensagem de início de seção de jogo para os demais. Essa mensagem será recebida e repropagada para os demais nós.
- **Fase 2 - Conexão individual**: Estado no qual as conexões de rede entre cada peer são criadas. Neste ponto que a rede P2P é criada. É importante que no final haja uma confirmação para saber se todos os nós foram enxergados pelos demais. Essa confirmação será o que definirá qual o próximo estado a ser executado.
- **Fase 3 - Estado de jogo**: Onde ocorre o jogo propriamente dito. É importante destacar que neste estado o módulo mantém-se o mais genérico possível, de forma que as informações a serem enviadas serão definidas pelo programador.

3.2.2 Transições de estado

Seguindo a esquematização em máquina de estados da seção anterior, precisamos definir com precisão as transições possíveis. Para deixar as descrições mais claras, será exemplificado como um nó qualquer da rede executa cada transição. Chamaremos esse nó de o jogador, embora a implementação do plugin seja feita de forma que a esconder essas transições do usuário. Nesse protocolo existem 4 transiçoes possíveis:

- **Fase 1** $\rightarrow$ **Fase 2**: Essa transição ocorre quando o jogador escolhe começar uma seção de jogo ou é recebida uma mensagem de início de jogo.
- **Fase 2** $\rightarrow$ **Fase 3**: Essa transição ocorre quando o jogador recebe uma mensagem de confirmação de que todos os nós da rede terminaram a fase 2 com sucesso.

- **Fase 2 → Fase 1**: Essa transição ocorre quando o jogador recebe uma mensagem de que algum outro nó sinalizando que não conseguiu terminar a fase 2.
- **Fase 3 → Fase 1**: Essa transição ocorre quando a condição de término da partida de jogo ocorre. Nesse ponto o jogador deve enviar uma mensagem final indicando aos demais que encerrou sua seção de jogo. Após essa mensagem ser enviada.

3.2.3 A terceira fase do protocolo

De todas as fases do protocolo, a única que precisa de uma discussão e explicação mais aprofundada é a fase 3. Isto porque um dos problemas mais cruciais de uma comunicação em P2P é garantir que todos os nós da rede terão uma informação confiável, e o mecanismo que fará isso está implementado na fase 3.

Por este trabalho tratar do problema específico de redes P2P em redes locais, não será necessária uma checagem quanto à procedência da mensagem, ainda mais por se tratar de jogos de partida rápida e com poucos jogadores. portanto o grande problema que temos de tratar é o saber se a mensagem está muito atrasada e é defasada em relação ao processamento sendo realizado na máquina do jogador.

O mecanismo adotado para resolver este problema foi o uso de um sistema de *timestamp*, isto é, para cada mensagem enviada será atribuído um carimbo de tempo. Dessa forma, uma mensagem só será aceita se estiver dentro de um intervalo de aceitação em relação à ultima mensagem enviada pelo jogador.

4 Implementação

4.1 Base da implementação

Como um dos objetivos deste trabalho é implementar um plugin para Unity 3D que crie conexão P2P entre jogadores, para isso foi escolhida a linguagem C# [9] [10] [12]. Para criar as conexões entre os peers foram utilizadas classes do módulo Sockets da plataforma .Net, seguindo como base a documentação contida na*webpage* oficial da Microsoft [13] e em fóruns oficiais da linguagem e da Unity [10] [11].

Em seguida, foi iniciada a criação de módulos básicos, responsáveis por encapsular as chamadas a sockets, suas inicializações, gerenciamento e operação. Isso facilita o processo de implementação do protocolo e diminui a complexidade do código. Foi criado um repositório no GitHub para controle de versão e backup do código que pode ser acessado em `https://github.com/renato-scaroni/mac499` (modificado pela última vez em 28/10/2014).

As classes básicas para a comunicação são:

- UDPBroadcast - Responsável por realizar envios de *broadcast*.
- UDPSendChannel - Responsável por realizar envios os envios das mensagens que não são de *broadcast*, além de guardar todas as informações relevantes para se enviar um pacote UDP para uma determinada porta
- UDPReceiveManager - Responsável por receber mensagem
- UDPListener - Responsável por guardar informações relevantes das conexões UDP para que seja feito o recebimento das mensagens

4.2 Organização do código

O código consiste de algumas classes base, como foi descrito no item anterior, cujo propósito é gerenciar tudo o que acontece com os sockets, sua criação e gerenciamento. O intuito disso é esconder ao máximo o que está acontecendo do ponto de vista da comunicação para que o programador se preocupe apenas em desenvolver sua lógica.

Toda execução do protocolo é gerenciada pela classe P2PNetwork. Essa classe implementa as fases do protocolo bem como gerencia suas transições e cria todos os canais de comunicação necessários utilizando as classes bases citadas. Essa classe herda de MonoBehaviour (ver seção 2.2.2 parágrafo 3) e deve ser atribuída a um GameObject.

Além da classe P2PNetwork, a classe UDPBroadCast também foi implementada como herança da classe MonoBehaviour pois ela envia mensagens com um intervalo pré determinado de tempo e para isso faz-se uso das medições de tempo do *game loop* da Unity 3D. Apesar de precisar ser associada a um GameObject, não é necessário que o programador o faça, pois a própria classe P2PNetwork se encarrega de iniciar um GameObject através da chamada de um método estático.

Classes que não tinham necessidade de controle de tempo ou fazem uso de processamento assíncrono, em especial as classes UDPListener e UPDReceiveManager, foram implementadas como classes padrões do C#, ou seja, herdando de Object apenas e utilizando threads. O gerenciamento das threads é feito através de uma thread chamada ManageListening da classe UDPReceiveManager.

Cada peer descoberto na rede será representado no programa como uma instância da classe KnownHost. Essa classe guardará se um host está ativo na rede, além de outras informações necessárias para o funcionamento do protocolo, como os sockets em que deve escrever as mensages durante a Fase 3 do protocolo. Um detalhe interessante dessa classe é que nela fica guardado um contador de tempo que mede há quanto tempo foi recebida a última mensagem vinda daquele nó. Porém a classe KnownHost não foi implementada com um MonoBehaviour, pois isso traria a necessidade de se instanciar um GameObject para cada peer conhecido, o que seria muito custoso computacionalmente. Alternativamente ela foi implementada como uma classe qualquer do C#. Suas instâncias são guardads em um hash na classe P2PNetwork e o controle do tempo de inatividade de cada peer fica a cargo dessa mesma classe.

Como mostrado até então, as classes processam em tempos diferentes, sendo que algumas possuem métodos que escutam assincronamente por mensagens, enquanto outras precisam fazer cálculos de tempo para rodar ciclos de vida ou executar tarefas agendadas. Então como que tudo se comunica e se sincroniza?

A resposta para essa pergunta é o sistema de eventos da Unity 3D. Esse sistema consiste em declarar um atributo estático que está associado ao pipeline da Unity e associar a ele um método do tipo delegate que atuará como *Handler* para quando esse evento for chamado. Outras classes podem associar métodos ao declarado junto com o evento para que sejam chamados quando isso acontecer. O evento só pode ser iniciado

por quem o criou, mas qualquer classe pode associar métodos próprios ao *Handler* de um evento.

Esse sistema é utilizado nesta implementação na classe de ReceiveManager para indicar que houve recebimento de mensagem. Na classe P2PNetwok é definido um método que vai lidar com esse evento quando ele ocorrer, ou seja, no caso ele vai decidir o que fazer com a mensagem recebida. Caso seja uma mensagem contendo um Alive, por exemplo, ou uma tentativa de abrir um canal direto de comunicação, o próprio método lida com ele, caso contrário lança outro evento para que um método definido pelo programador possa pegar a mensagem e fazer uso dela.

4.3 Alguns detalhes importantes

Agora que já temos uma noção de como o código foi organizado e como ele se interliga, é importante vermos alguns detalhes que ficaram vagos na descrição do protocolo e que foram definidos na implementação. Após alguns testes preliminares com o que já foi implementado, ficou definido que o tempo de vida de um peer é de 15 segundos, isto é, caso não seja recebido um Alive deste neste intervalo de tempo, o peer é marcado como inativo e não será levado em conta na formação de uma seção de jogo além de não aparecer na lista de hosts ativos na rede.

Outro fator importante é o tempo de intervalo entre o envio das mensagens de Alive. Na implementação atual este tempo ficou difinido como sendo de 2 segundos, por gerar uma quantidade de pacotes na rede suficiente para suprir perdas, mas que não congestiona o fluxo quando houver situação com grande número de jogadores.

Os testes referidos nos parágrafos acima foram realizados em 2 cenários: um de rede wifi pequena e outro na rede Linux do IME-USP. No caso da rede wifi foram utilizados um roteador tp-link WR841N e um notebook com placa rede wifi Qualcomm Atheros AR9485WB-EG, processador core i7 e 8Gb de memória RAM com Windows 8.1 rodando 3 máquinas virtuais pelo Virtual Box conectadas a rede em modo bridge, cada uma rodando Linux Ubuntu 14.10.

Já no caso da rede Linux o teste foi feito nas duas salas existentes atualmente em até três máquinas diferentes. Todas utilizando placas de rede gibabit, sendo as da sala BCC com processador AMD Phenom II X6 e 4 Gb de memória RAM e os da outra sala com processador Intel Celeron e também 4 Gb de memória RAM. Chegaram a ser realizados alguns testes entre as salas e com máquinas na mesma sala.

Também ficou definido um padrão para as mensagens enviadas. Sempre as mensagens serão strings e terão a seguinte formato (sendo o caracter \t (tabulação) sendo usado como um token delimitador das partes da mensagem):
IP_DO_EMISSOR\tTIPO\tMENSAGEM\tPORTA_PARA_ONDE_RESPONDER

4.4 Desafios

Durante o processo de implementação foram encontrados diversos desafios de implementação, em especial na integração da Unity. Por exemplo, o fato de existirem threads do plugin paralelas executando gerava um problema quanto a sincronização do programa, uma vez que algumas partes seguirão o *game loop* e outras executarão paralelamente, em especial as threads que cuidam de escutar mensagens para um socket. Esse problema foi resolvido com o uso de eventos da Unity que cuidam de avisar as threads que executam respeitando o *game loop* que alguma thread paralela recebeu uma mensagem ou preparou algum dado para ser usado.

Ainda sobre threads paralelas, houveram vários problemas relacionados a escuta e troca de mensagem assíncrona via sockets, uma vez que a Unity não dá suporte total ao .NET(a Unity em C# é baseada na plataforma Mono, que é um subset, ou seja, uma versão reduzida do .NET). Com isso, a alternativa foi criar métodos que escutassem síncronamente escutando em threads paralelas e controlar a execução dessa thread através do ReceiveManager. Para isso foi preciso utilizar um mutex para sincronizar a execução e garantir que sempre que necessário haveria apenas uma thread escutando cada socket.

Além disso, ainda relacionado ao problema da Unity não dar suporte total ao .NET, houve o problema de definir a máscara de sub-rede e o endereço de *broadcast* para enviar os dados aos demais nós da rede. Para resolver esse problema foi criado um método na classe responsável por gerenciar as interfaces de rede numa máquina (InterfacesManager) que realiza um cálculo em cima do primeiro octeto do IP associado a cada interface para descobrir sua máscara de sub-rede. É importante ressaltar aqui que como o foco desde o início foi fazer um plugin focando num cenário de jogos em rede local, dificilmente uma máscara de rede fugirá do padrão citado no último parágrafo da seção 2.1.1. Abaixo está reproduzido o código usado para fazer esse cálculo:

```
static uint  ReturnFirtsOctet(string ipAddress)
{
        IPAddress iP = System.Net.IPAddress.Parse(ipAddress);
        byte[] byteIP = iP.GetAddressBytes();
```

```
        uint ipInUint = (uint)byteIP[0];
        return ipInUint;
}

static string ReturnSubnetmask(String ipaddress)
{
        uint firstOctet =  ReturnFirtsOctet(ipaddress);

        if (firstOctet >= 0 && firstOctet <= 127)
                return "255.0.0.0";
        else if (firstOctet >= 128 && firstOctet <= 191)
                return "255.255.0.0";
        else if (firstOctet >= 192 && firstOctet <= 223)
                return "255.255.255.0";
        else return "0.0.0.0";
}
```

4.5 API e integração com projetos da Unity 3D

Todo o processo de implementação foi pensado de forma a ter um resultado final que fácil de integrar com qualquer projeto na Unity. Por isso, para utilizar o plugin basta atribuir a um GameObject qualquer (pode ser inclusive um que não esteja de fato associado a um objeto visível em cena ou com nenhum outro script associado). Apenas isso fará com que o protocolo já funcione.

A interface de um programa qualquer com o plugin se dará através dos eventos da Unity, ou seja, basta que sejam definidos dois métodos: um que tratará as mensagens vindas das mensagens recebidas na terceira fase e outro para decidir o que fazer caso a fase 2 dê errado e o programa retorne para a fase 1. Além disso, para fazer as transições de fase basta chamar os eventos de começar jogo, para ir da fase 1 para a três (as transições para a fase dois e para sair da fase 2 serão feitas automaticamente) e terminar jogo para voltar a fase 1.

5 Conclusão e considerações finais

De acordo com os objetivos do trabalho, que consistia em propor um protocolo, testar a viabilidade de sua implementação e integração com a Unity e gerar um plugin para integração com a mesma, podemos dizer que o trabalho foi satisfatório.

Como planos futuros para este trabalho temos a integração deste plugin com alguma aplicação desenvolvida na Unity, de preferência o jogo shuttles. Este jogo foi desenvolvido por mim e, por ser um jogo casual, de partida rápida e dinâmica, seria uma excelente oportunidade de testar mais profundamente e aprimorar o plugin desenvolvido.

Bibliografia

[1] Computer networking: A Top-Down Approach [Livro]. KUROSE, ROSS. 6a edição. Editora Pearson

[2] Definição das sete camadas do modelo OSI e explicação de suas funções [Internet]. Página de suporte da Microsoft. [última atualização em 29 de novembro de 2013, citado em 01/10/2013]. [último acesso em 28/11/2014]. Disponível em: http://support.microsoft.com/kb/103884

[3] RFC: 791 - INTERNET PROTOCOL [Internet]. Página de descrição do IPv4. [último acesso em 28/11/2014]. Disponível em: https://tools.ietf.org/html/rfc791

[4] Internet Protocol, Version 6 (IPv6) Specification [Internet]. Página de especificação do IPv6. [último acesso em 28/11/2014]. Disponível em: www.rfc-base.org/rfc-2460.html

[5] IP Version 6 Addressing Architecture [Internet]. Página de descrição da arquitetura do IPv6. [último acesso em 28/11/2014]. Disponível em: http://www.rfc-base.org/rfc-4291.html

[6] Internet Standard Subnetting Procedure [Internet]. Página de especificação de uma sub-rede. [último acesso em 29/11/2014]. Dísponível em: http://www.rfc-base.org/rfc-950.html

[7] Redes e Servidores Linux [Livro]. MORIMOTO. 2a edição. Editora Sulina. Dísponível em versão online gratuita em: http://www.hardware.com.br/livros/linux-redes/ [último acesso em 29/11/2014]

[8] Unity Manual [Internet]. Manual da Unity 3D. [último acesso 29/11/2014]. Disponível em: http://docs.unity3d.com/Manual/index.html

[9] Unity Scripting Reference [Internet]. Página de documentação da API da Unity. [último acesso 29/11/2014]. Disponível em: http://docs.unity3d.com/ScriptReference/index.html

[10] Unity Comunity [Internet]. Página do fórum da comunidade da Unity. [último acesso 29/11/2014]. Disponível em: http://forum.unity3d.com/

[11] Unity answers [Internet]. Página de dúvidas da Unity. [último acesso 29/11/2014]. Disponível em: http://answers.unity3d.com/

[12] Referência de C# [Internet]. Documentação oficial da linguagem. [último acesso 29/11/2014]. Disponível em: http://msdn.microsoft.com/pt-br/library/618ayhy6.aspx

[13] Biblioteca de classes .NET Framework [Internet]. Documentação oficial do .NET [último acesso 29/11/2014]. Disponível em: http://msdn.microsoft.com/pt-br/library/gg145045%28v=vs.110