Construção de aplicação P2P para streaming de Áudios MP3

Guilherme Figueiredo Terenciani[1], **Maximilian Jaderson de Melo**[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79.070-900 – Campo Grande – MS – Brasil
Mestrado Acadêmico em Ciência da Computação
Disciplina de Redes de Computadores

{guilherme.terenciani, maximilian.melo}@ifms.edu.com

1. Introdução

Aplicações de *streaming* de áudios e filmes tem se tornado cada vez mais comum ao longo dos anos. Aplicações como Netflix ®, Youtube® e Spotfy® são os grandes exemplos de aplicações que ganharam espaço com o avanço da internet.

Buscando compreender o funcionamento arquiteturas de redes P2P(*Peer to Peer*) e softwares de *streaming* como os supracitados, este trabalho objetiva a construção de um software na linguagem de programação Python que implementa algumas das funções de programas de transmissão de dados multimídia. Para mais detalhes sobre arquiteturas de redes, veja a Seção 2, mais especificamente a Subseção 2.2.

Neste trabalho foi criado um programa capaz de descobrir os *peers* conectados na mesma rede local (centralizada) e descobrir os arquivos mp3 que hospedados na mesma. Após a descoberta dos arquivos pela aplicação, pode-se solicitar aos *peers* que enviem o arquivo para ser executado por meio do protocolo UDP.

2. Fundamentação teórica

Uma arquitetura de rede descreve um conjunto de camadas e protocolos, além de a forma como os *hosts* interconectam-se. As duas arquiteturas mais populares são a Cliente-Servidor e P2P, cada qual contendo vantagens e desvantagens para cada cenário de utilização. As arquiteturas são descritas a seguir.

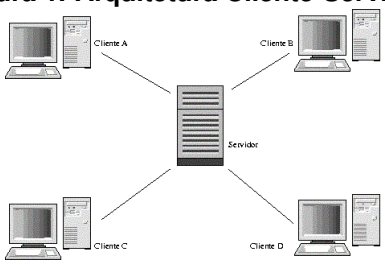
2.1. Cliente-Servidor

Na arquitetura Cliente-Servidor há sempre a figura de dois elementos um *host* provê um determinado serviço (servidor) e um ou mais *hosts* consumidores do serviço (cliente(s)). É uma arquitetura que centraliza o serviço em um único *host*. As aplicações web são exemplos cotidianos do emprego dessa arquitetura. Diversos outros serviços comumente a empregam: FTP, Telnet, E-mail, etc [1]. Dentre as diversas vantagens e desvantagens, pode-se citar a simplicidade de implementação/segurança e a fragilidade do serviço (o servidor é o elo fraco, que compromete o serviço em casos de indisponibilidade), respectivamente. A arquitetura é representada na Figura 1.

2.2. P2P

A arquitetura P2P (*Peer to Peer*) consiste na dependência mínima de servidores dedicados. A arquitetura explora a comunicação direta entre os *hots*, chamados de *peers*. Os *peers* não são propriedade de um provedor específico, mas computadores pessoais conectados, eventualmente, à rede [1]. Esse tipo de arquitetura garante robustez, permite que

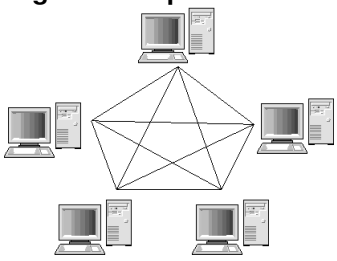
Figura 1. Arquitetura Cliente-Servidor



Fonte: GTA UFRJ

qualquer *peer* provenha serviços aos demais e redundância de dados, entretanto demanda maior infraestrutura para funcionar. Diversas aplicações empregam a arquitetura, como: serviços de transferências de arquivos, aceleradores de download, telefonia pela Internet, IPTV, etc. Na Figura 2 é ilustrado uma arquitetura P2P.

Figura 2. Arquitetura P2P



Fonte: GTA UFRJ

3. Funcionamento do Programa

3.1. Linguagem de Programação e Requisitos

Para a construção da aplicação foi utilizada a linguagem de programação Python em sua versão 3.6 e foi testado apenas para o sistema operacional Linux.

O programa pode ser executado em qualquer máquina, não obrigatoriamente com alto poder computacional e necessita a instalação de alguns pacotes como:

- pip install pydub
- sudo apt-get install python3-pyaudio
- sudo apt-get install ffmpeg
- pip3 install numpy
- python3 -mpip install matplotlib
- pip3 install pyqt5
- pip3 install matplotlib
- pip3 install pandas

O código fonte foi versionado utilizando a ferramenta GIT e se encontra no repositório `https://github.com/guilhermeterenciani/teretorrent`.

Na Figura 3 é mostrado os passos para a execução do software. Devemos apenas instalar os requisitos da aplicação e rodar utilizando a linguagem python. Para usuários com mais interesse, ainda podemos mudar o valor de RTT no arquivo RTT.TXT e mudar o valor de F na classe randomDelay linha 15. Também podemos mudar o tamanho do buffer na linha 61 da classe Torrent.

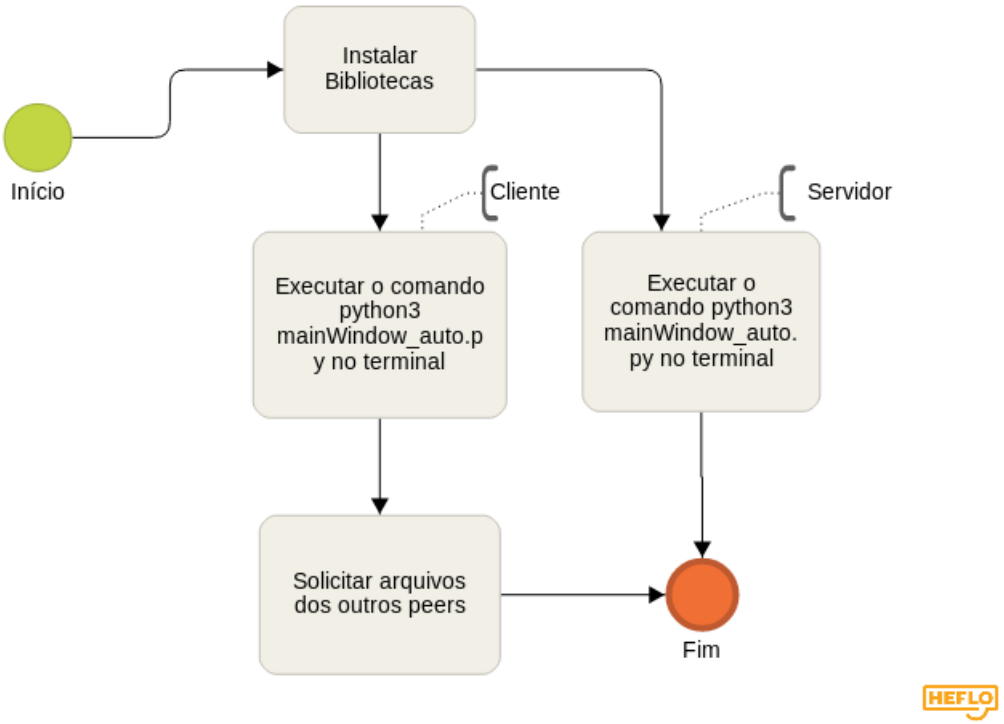


Figura 3. Esboço inicial

3.2. Classes

Na Figura 4 é mostrado o funcionamento geral do software de *Streaming* MP3 criado para o trabalho da disciplina de redes de computadores. O software é dividido em algumas partes e suas principais funcionalidades são mostradas na figura.

Abaixo é mostrada a listagem de classes e uma breve descrição de suas funcionalidades para o programa.

1. **Ui_MainWindow**
   A classe *Ui_MainWindow* é responsável pela criação da interface gráfica de usuário (GUI). Para o *design* da GUI foi adotado *QT Framework*[1] devido à sua praticidade, extensa documentação e comunidade ativa. O QT é originalmente voltado ao desenvolvimento de aplicações robustas em C++, sendo responsável por exemplo pela criação do ambiente gráfico KDE[2], popular na comunidade Linux. Apesar de ser voltado para uma linguagem diferente, é possível aproveitar de seus diversos recursos para *design* de GUI's (*QT Designer*) por meio do empacotador do QT chamado PyQt[3].
   O objetivo da GUI desenvolvida é focado na facilidade de uso e simplicidade. Dessa maneira, o *wireframe* ilustrado na Figura 5 foi desenvolvido.
   Em seguida, resumidamente, no projeto da interface gráfica são adicionados apenas um componente de tabela (QTableView) e um botão (QButon), todos conectados e gerenciados por meio dos eventos[4]. O resultado da GUI pode ser visualizado

[1] https://www.qt.io/
[2] https://kde.org/
[3] https://wiki.python.org.br/PyQt
[4] https://www.riverbankcomputing.com/static/Docs/PyQt4/qevent.html

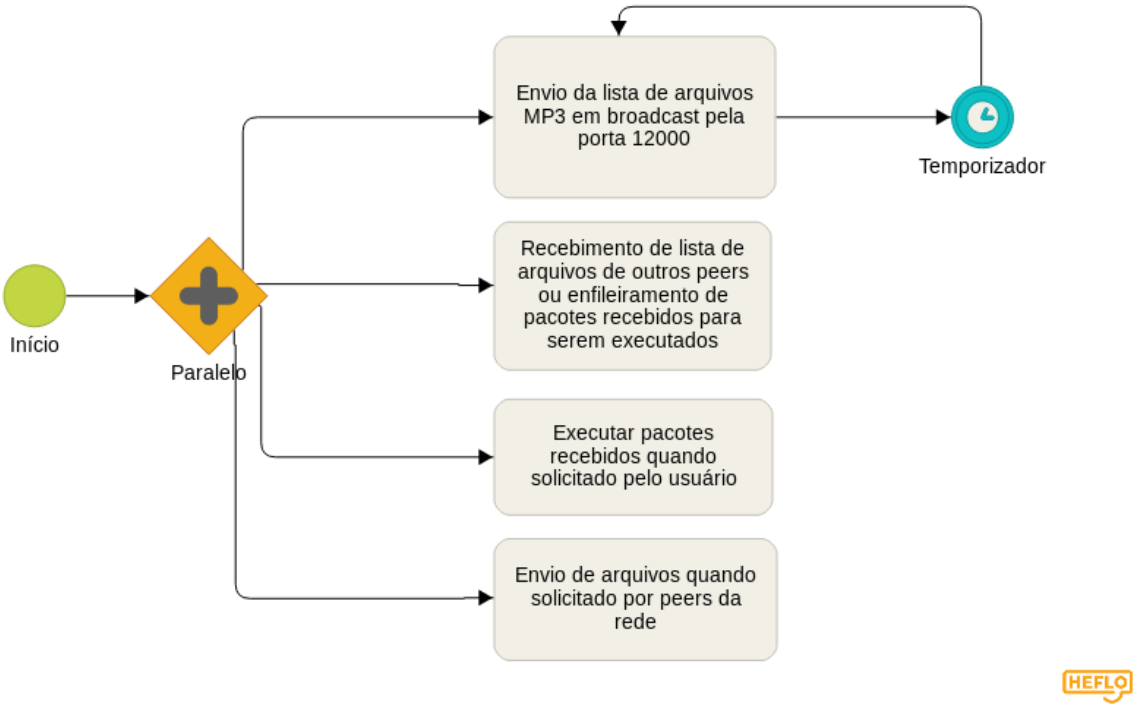


Figura 4. Funcionamento geral do software

na Figura 6

Para a realização da listagem, é necessária a conversão dos dados advindos da classe Torrent num formato compatível com a classe *QListView*. Para esta funcionalidade, é desenvolvida a classe *MyTableModel*, capaz de listar os *Sedeers* e seus arquivos. A cada 3 segundos, a listagem de arquivos na GUI é atualizado por meio da classe *TableModelUpdater*.

2. **MyTableModel**
   Esta classe é responsável pro criar um modelo de tabela, descrevendo o cabeçalho da tabela, as linhas (conteúdo da tabela), além de oferecer a possibilidade de tratar eventos de manipulação dos dados, ou permitir seu tratamento por meio externo. Esse modelo é responsável por armazenar uma cópia da listagem dos arquivos nos *sedeers* e IP's dos computadores que detém cada arquivo.
3. **TableModelUpdater**
   Esta classe é resumidamente uma *Thread* trabalhadora, destinada a monitorar mudanças na lista de arquivos da classe *Torrent* e emitir um sinal de atualização da lista de *peers* para *MyTableModel*. A classe é desenvolvida em cima da *QThread*. A atualização da lista se faz necessário via esquema de emissão e captura de sinais devido às políticas de encapsulamento e proteção de elementos gráficos herdados de *QWidget* (raiz dos componentes gráficos) do QT.
4. **RandomDelay**
   Esta classe se dedica única e exclusivamente a duas tarefas: simular um atraso (*delay*) na rede e decidir se um pacote x deve ou não ser recebido.
   O *delay* toma como entrada um arquivo de texto "rtt.txt"contendo um valor definido para o RTT do sistema enquanto o valor da média E[X] foi fixado como o valor 100, constante. Também é obtido como parâmetro de entrada, o número

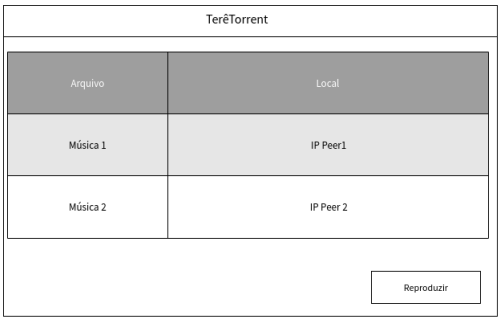


Figura 5. Esboço inicial

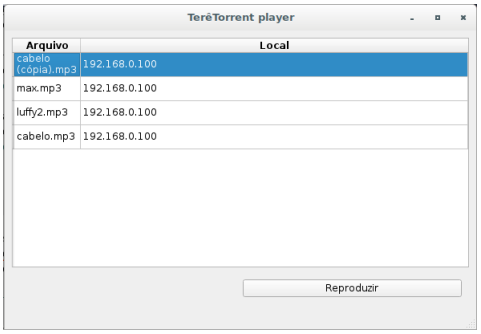


Figura 6. Aparência da GUI

de pacotes da amostra. O cálculo do atraso é feito considerando uma distribuição exponencial, conforme mostrado na Equação 1.

$$P[a < x < b] = -\frac{1}{\lambda}.e^{u}, \tag{1}$$

na qual u é definida em 2 como uma distribuição uniforme (pela biblioteca *numpy*) com função de densidade e probabilidade definida abaixo.

$$u[x] = \frac{1}{b-a}, \tag{2}$$

com $a = 0$ e $b = 0.1$, definidos empiricamente. O número de elementos de u é definido como o número de pacotes total a transmitir.
A aplicação da Equação 1 resulta de um arranjo de valores discretos, aos quais é adicionado a metade do valor do RTT, formando assim um arranjo de valores de atraso, acessadas posteriormente no módulo de atrasos.
Nesta classe também é implementada uma política para decisão de descarte ou recebimento do pacote atual, representada no Algoritmo 1.
O valor de $f = 0.1$ (10%) indica que serão rejeitados uma pequena parcela dos pacotes.

5. **Torrent**
   A classe Torrent é o coração do sistema. Ela é responsável por tocar os arquivos recebidos dos Sedeers e também responsável por enviar os arquivos que contém em sua máquina quando for solicitado por por outro peer.

Algoritmo 1: Política de descarte de pacotes

```
1 pEntrega = 1 − np.random.rand()
2 if pEntrega > f then
3 |   entrega pacote
4 else
5 |   descarta pacote
6 end
```

A classe Torrent contém vários métodos importantes para o pleno funcionamento do sistema, e alguns desses métodos acabam tendo que rodar em threads diferentes pelo fato da classe funcionar como um cliente e servidor ao mesmo tempo. Abaixo são listados e explicados alguns dos métodos mais importantes dessa classe:

- **enviaArquivos**
  – Ao ser instanciado um objeto do tipo Torrent pela nossa interface gráfica. O construtor da linguagem inicia para a nossa aplicação duas Threads que rodarão em paralelo com nossa aplicação e funcionaram como os clientes e servidores da nossa aplicação. Essas duas threads são denominadas enviaArquivos e recebeArquivos. A threads enviaArquivos é responsável por a cada 1 segundo enviar em broadcast todos os arquivos MP3 que a aplicação detém em sua pasta **sender**, que fica dentro da aplicação.
- **recebeArquivos**
  – A thread recebeArquivos é responsável por controlar a aplicação e executar ações baseadas na chegada de pacotes pela rede.
  Existem basicamente em nossa aplicação dois tipos de pacotes. E para a construção do pacote foi utilizada a biblioteca python Pickle. O módulo pickle implementa um protocolo binário para serialização e desserialização de objetos em Python. Nós criamos uma lista em python e adicionamos os dados que precisamos enviar para o lado do cliente e então utilizamos o modulo pickle para converter a estrutura em bytes. E do lado do cliente conseguimos fazer o caminho inverso e refazer o objeto python passado pelo servidor.
  Existem basicamente dois tipos de pacotes. Pacotes de controle e pacote de transferência de arquivos. Os pacotes de controle respeitam o seguinte padrão demonstrado na Tabela 1. No campo Tipo pacote ele pode conter quatro valores, de zero a três. Se contiver o valor 0 significa que o pacote enviado é um pacote de informações de lista de arquivos que algum peer e que foi enviado por broadcast; Se contiver o valor 1, significa que o pacote é um arquivo de requisição de download por algum peer da rede. E que no campo data temos o nome do arquivo que deve ser enviado a esse peer; Se contiver o valor 3, significa que o pacote é uma solicitação de pacotes faltantes de um arquivo MP3 que um peer necessita sobre algum arquivo. O programa deve então enviar para o peer os pacotes que o mesmo solicitou; e por último, quando o campo Tipo Pacote contém o valor 2, significa que o dado recebido é um pacote solicitado e que deve ser encaminhado para ser tocado pela própria máquina. Ou seja, algum peer da rede enviou o pacote para a nossa aplicação esperando que a mesma a execute.

Tabela 1. Pacotes de controle de dados

bytes	0 ——————-31	32——————-63
0	Tamanho do Objeto Lista	
64	Tipo Pacote	DATA
...		
3553		

Pacotes que detém o valor de Tipo Pacote com o valor 2 tem uma estrutura um pouco quanto diferente dos demais. Na Tabela 2 pode ser observado a diferença entre os pacotes de controle. No pacotes de áudio temos o valor da sequencia que está sendo enviado e a quantidade de pacotes que o arquivo de áudio.

Tabela 2. Pacotes de som

bytes	0 ——————-31	32——————-63
0	Tipo Pacote	Número Pacote
64	Quantidade Pacotes	DATA
...		
3553		

A thread recebeArquivos é responsável por receber todos os arquivos que chegam na aplicação. Quando chega requisições de arquivos que estão na máquina local ela é responsável por iniciar a threading de envio de arquivos para um peer da rede e quando recebe arquivos de áudio, ela é responsável por armazenar os dados em uma fila ordenada para que possam ser executados pela thread **play**.

- **requisicaodeArquivo**
  – A classe Torrent fornece um método chamado requisição de arquivos, este método é responsável por receber o pedido da interface gráfica e e pedir a lista de servidores o arquivo que deve ser executado pela método **play**. Este método é responsável criar um pacote de controle e enviar para cada servidor um pedido de requisição de arquivo. A partir do envio do pacote de solicitação o método de requisição de arquivos inicia a thread de play que será responsável por tocar os pacotes que estão na fila de reprodução.
- **play**
  – A thread play é iniciada a partir de uma requisição de arquivos aos peer que estão na rede. Ela é responsável por pegar os dados que foram enfileirados pela threading **recebeArquivos** e executados para que possam ser ouvidos pelos usuários da aplicação.
  Diferentes políticas de execução foram implementadas. A primeira, o player ao se deparar com uma falha de perda excessiva de pacotes parava por 1 segundo e tentava voltar a executar sua aplicação. E a segunda, que faz mais sentido em um player de áudio, foi a criação de um buffer de dados que pode ser fixado seu tamanho no início da aplicação e quando há uma perda excessiva de pacotes a aplicação para de executar e carrega o buffer de dados. O buffer de dados pode ter tama-

nho váriado, e devido a perda de pacotes pelo módulo de perdas, foi criado um método de requisição de arquivos faltantes que o buffer acha que foram perdidos, ou estão muito atrasados. O método criado foi o "**requisicaodeArquivosFaltantes**"e com ele é possível pedir a um ou mais senders que envie pacotes que foram perdidos ou estão atrasados.
A politica implementada é que quando existe mais de um sender é que podemos pedir diferentes pacotes para diferentes peers. Algumas vezes acabamos repetindo pedidos de envios para dois peers mas, devido a perda e indisponibilidade de envios de pacotes, pacotes duplicados podem e devem ser utilizados pela nossa aplicação, a fim de conseguir ter um desempenho melhor.

6. **ModuloAtraso**

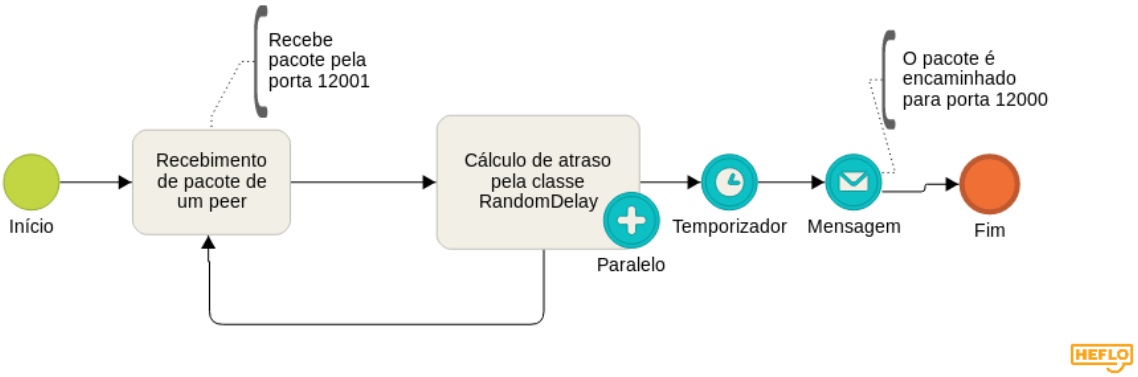


Figura 7. Funcionamento da classe de módulo de atraso

–Na Figura 7 podemos ver um diagrama de sequência do funcionamento da classe de módulo de atraso. Nela podemos ver como a classe de módulo de atraso cria os atrasos gerados pela rede de computadores. O funcionamento da classe de Módulo de atraso pode ser alterado na classe Torrent de forma automática, mudando apenas o valor de utilização da variável MODULO_ATRASO para falso. Quando o valor da variável está setada como verdadeiro, todos os dados da aplicação são enviados para a porta 12001 dos outros peers. A classe de Módulo de atraso é responsável por escutar essa porta e a partir do calculo feito pela classe RandomDelay entregar o pacote na porta 12000 com o atraso criado pela aplicação.
A classe Módulo de atraso recebe todos os pacotes de áudio que são trocados pelos peers da aplicação e cria uma thread onde é passado um parâmetro que é a quantidade de tempo que deve ser esperado para um pacote ser entregue para a aplicação. O cálculo de perda ou envio do pacote e a quantidade de tempo que um pacote deve esperar para ser entregue é feito pela classe RandomDelay. A classe Módulo de atraso instância um objeto da classe RandomDeley e o utiliza para calcular os atrasos e perdas dos pacotes.

4. Resultados

Os resultados da aplicação são satisfatórios, na Figura 8 vemos a aplicação respondendo a uma solicitação de envio de arquivos realizada por um peer da nossa rede P2P. Ao receber a solicitação o software identifica que pode enviar o arquivo para o peer solicitante.

A nossa primeira implementação segue as recomendações impostas pela professora desta matéria e respeita a rega de disponibilidade de pacotes, que ao se deparar com uma indisponibilidade aguarda 20ms e consulta se pode enviar o pacote. A aplicação envia um pacote a cada 20ms e ainda aguarda 20ms quando o pacote não está disponível. Como a indisponibilidade é na casa de 20% e o gráfico tem aproximadamente 28000 pacotes, não conseguimos identificar essas indisponibilidades pelo gráfico gerado.

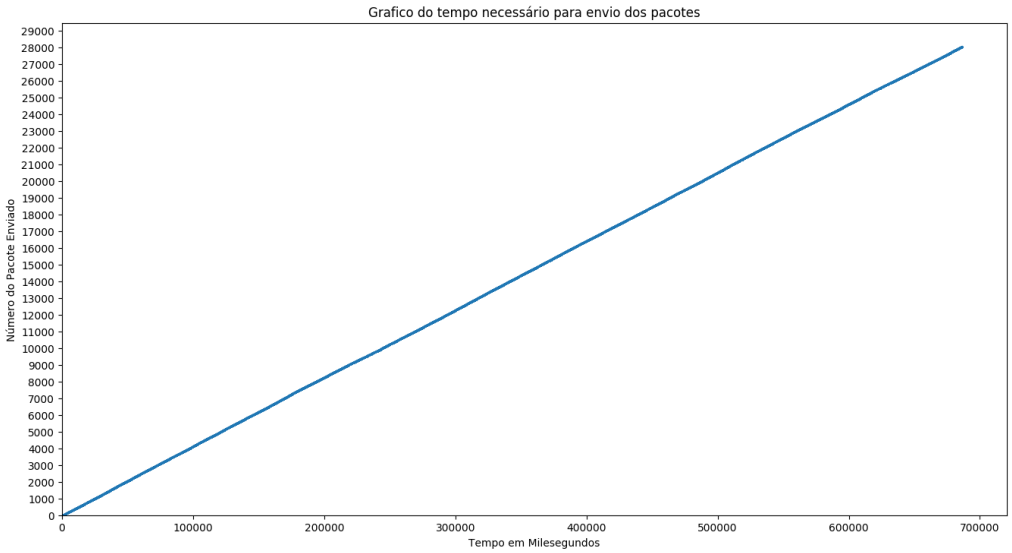


Figura 8. Funcionamento da aplicação enviando arquivos para um peer solicitante

Na figura 8 a resposta a uma solicitação de peer é feita de forma sequencial e cada pacote é enviado com um intervalo mínimo de 20ms.

O peer de envio mostrado na Figura 8 foi utilizado com semeador da aplicação para uma solicitação de arquivos descrito na Figura 9. A politica implementada para o primeiro receptor de arquivos era que ao se deparar com um esvaziamento do buffer de recebimento de dados o mesmo ficaria parado por 1 segundo até que pudesse voltar a tocar os dados que foram recebidos. Essa politica implementada fica evidente pelo gráfico da Figura 9 observando os dados dos pacotes que foram tocados. Na figura conseguimos ver que o gráfico dos pacotes tocados assume uma forma de escada em que na parte que é horizontal são os momentos em que o player fica parado aguardando o recebimento de mais pacotes para poder executar.

Está primeira implementação é interessante e foi muito importante para o pleno entendimento do trabalho, nela podemos entender o funcionamento dos seeders entender como nossa aplicação pode encontrar falhas e atrasos na rede.

A primeira implementação apresenta diversas desvantagens em aplicações de compartilhamento de áudio na rede, pois além de ficar muito tempo ociosa, a aplicação ainda apresenta muita perda de pacotes. Essas desvantagens são responsáveis por transmissão muitas vezes intermitentes no receptor, que pode ficar falho e não transcorrer de forma suave.

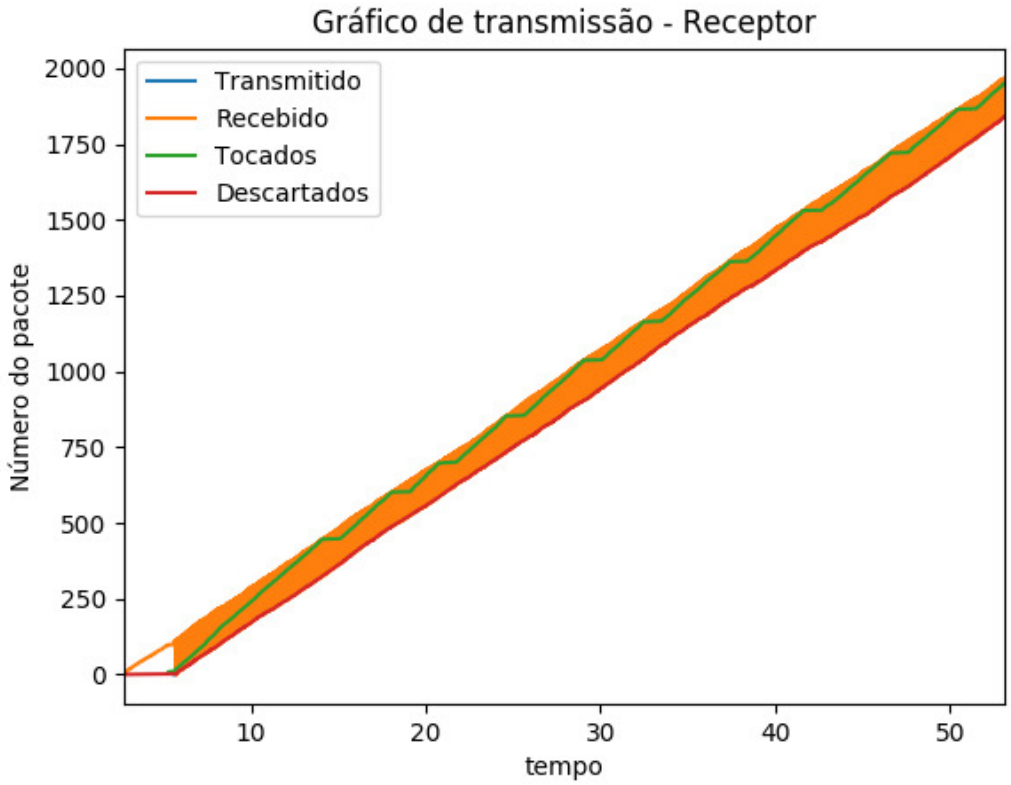


Figura 9. Aplicação funcionando com a primeira política implementada: Ao não existirem pacotes no buffer de execução a aplicação para por 1segundo

Sobre a segunda aplicação, foi criada uma aplicação capaz de executar os arquivos que chegam do lado de receptor e aguardar o carregamento do buffer quando os dados da aplicação ainda não chegaram. A Figura 10 descreve o funcionamento da aplicação com dois seeders enviando arquivos para a máquina solicitante de arquivos, com um buffer de 2% do tamanho do arquivo uma taxa f de 0.1 e um rtt de 0.001. Nessas condições é possível ver duas linhas no gráfico que são os dados recebidos. Os dados recebidos de um *seeders* e depois pelo outro. Como os dois são sequenciais não faz muita diferença para a aplicação ter vários *seeders*. Única melhora considerável é que ao perder arquivos vindos do primeiro *seeder* que envia os pacotes, o outro *seeder* pode enviar um pacote que o primeiro perdeu. Fazendo com que a aplicação não tenha muitos dados perdidos pela taxa f.

Na Figura 11 podemos destacar a essência da aplicação. Devido ao arquivo de teste ser extenso, a aplicação não se comporta eficientemente com o modelo aleatório. De modo a ganhar com os benefícios da transmissão aleatória, foi criado um módulo que verifica quando há falhas no *buffer* por falta de dados e alimenta o mesmo de forma semi aleatória. Foi testada duas politicas para o carregamento de elementos no *buffer* próximo ao momento da reprodução: política sequencial e aleatória. Esse carregamento por requisições de pacotes que estão próximos a execução, não tem nada em comum com as duas filosofias de envio de arquivos. O carregamento do *buffer* criado neste trabalho faz requisição dos arquivos que estão próximos a execução a fim de conseguir recuperar

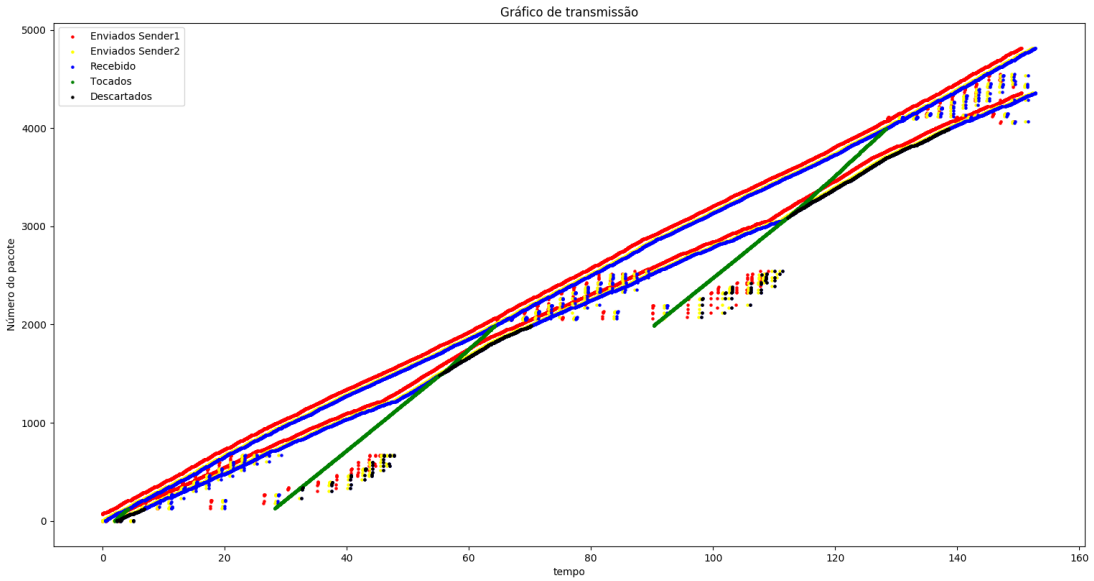


Figura 10. Funcionamento da aplicação no recebimento de um arquivo MP3 de dois seeders sequenciais

o *player*, visando atenuar o impacto de atrasos de reprodução prolongados na aplicação. Logicamente, o tempo de carregamento do *buffer* é penalizado e a aplicação acaba reenviando vários pacotes que já seriam enviados posteriormente. Mas, a fim de tentar recuperar o *player* no menor tempo possível é aceitável essa demora. Vale destacar na, Figura 11, que nossa forma de aceleração se inicia próximo ao instante de 10ms e requisita os pacotes faltantes para que nossa aplicação possa executados. A aplicação só sai desse modo de recuperação quando chegam todos os N pacotes sucessores ao último pacote tocado pelo *player*, sendo N pacotes o tamanho do *BUFFER*.

Na figura 11 temos uma diferente configuração para o módulo de atrasos. Fixamos o RTT em 1 segundo mais uma perda de pacotes de aproximadamente 50% com um *buffer* de 5% do arquivo. Nessas condições conseguimos executar a nossa aplicação. Mas, contudo, a aplicação demorou aproximadamente 80 segundos para conseguir executar o primeiro bloco de som. O tamanho do arquivo de teste e a dificuldade imposta pelo módulo de atraso fazem nossa aplicação sofrer nessas condições. Quando aumentamos o tamanho do *buffer* com uma perda de pacote tão alta, nosso módulo de recuperação rápida de falhas no *player* acaba sufocando os *seeders* por pedidos de arquivos faltantes. Em alguns casos ele acabou estourado o uso da memória de alguns *seeders*. Isso se deu ao fato de que a cada nova requisição de arquivo, nos utilizamos a biblioteca python Audio_Segment para carregar o arquivo de áudio na memória. Como a taxa de perda de pacotes é muito alta, o peer envia muitas solicitações para seus *seeders*. E nas configurações de teste, na qual as máquinas tinham apenas 2GB de memória, foi o suficiente para fechar a aplicação.

Finalmente, na Figura 12 temos a configuração do módulo de perdas de pacotes e atrasos configurados com um f de aproximadamente 20%, um buffer de 5% e um RTT de aproximadamente 2 segundos. Na Figura 12 é claramente visível a demora para a aplicação receber os dados do lado do cliente. Se olharmos a região mais à esquerda, podemos perceber que cada marcação amarela ou vermelha que são os dados enviados

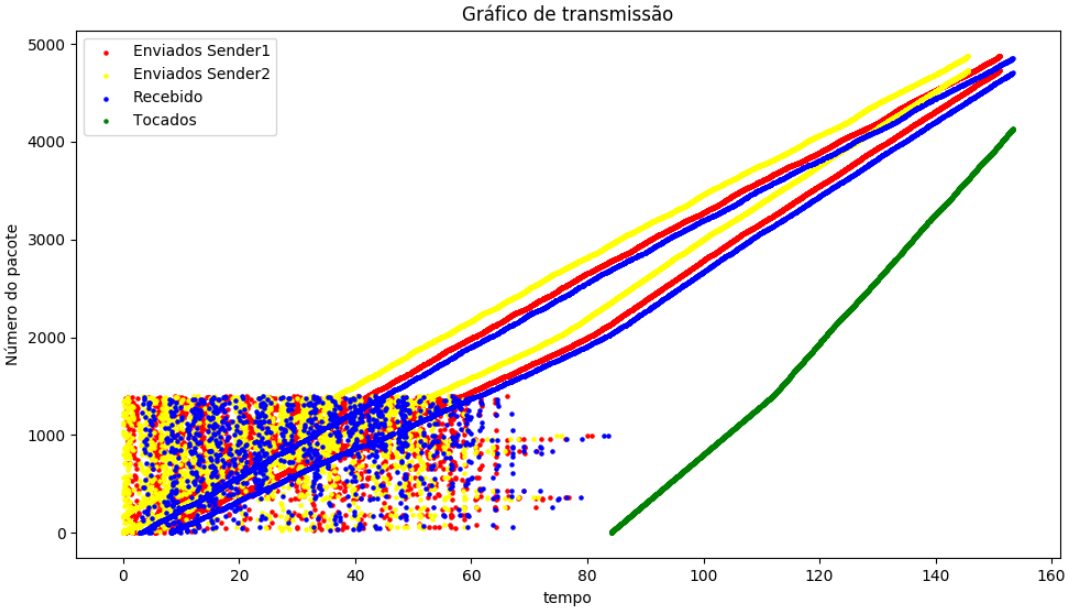


Figura 11. Funcionamento da aplicação com uma taxa de perda de pacotes de 50%

pelos *seeders* demoram a se tornar azul que é quando realmente chegam na aplicação.

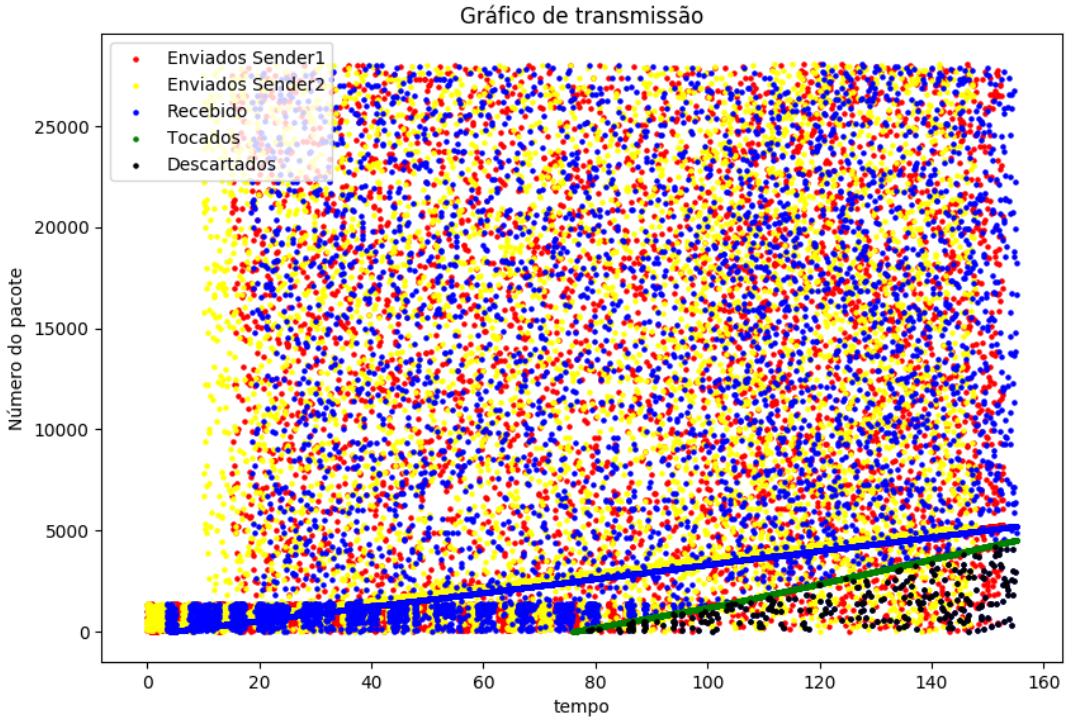


Figura 12. Funcionamento da aplicação com uma taxa de perda de pacotes de 20% e um seeder aleatório e um sequencial com RTT de 2segundos

5. Conclusão

Este trabalho teve seu propósito alcançado. Com a implementação dos *seeders* e receptores é possível representar o funcionamento de uma arquitetura p2p, além de claro, exemplificar e mostrar a importância dos *buffers* no lado do receptor.

Apesar de todas as funcionalidades apresentadas, há espaço para melhorias, como: aprimorar o algoritmo de recuperação de falhas rápidas, além de melhorar a utilização dos *seeders* aleatórios, quando operando em modo aleatório.

Referências

[1] James Kurose and Keith Ross. Computer networks: A top down approach featuring the internet. *Peorsoim Addison Wesley*, 2010.