T2 - Laboratório de Redes e Computadores

Guilherme S. Rizzotto[1], João F. Leão[2]
Escola Politécnica— PUCRS

06 de junho de 2020

Resumo

O trabalho presente serve como método de avaliação da disciplina de Laboratório de Redes de Computadores e Redes de Computadores ministrada pela professora Cristina Moreira Nunes como requisitos para a obtenção do bacharelado em Engenharia de Software. O trabalho envolve a implementação de uma aplicação com o protocolo UDP que simule o comportamento de um protocolo de transferência de arquivos orientado a conexão.

1. Introdução

1.1. Visão Geral

Durante o quinto semestre do curso de Engenharia de Software, as disciplinas relacionadas ao conteúdo de redes apresentaram diversos novos conceitos e revisitaram alguns já estudados em diferentes cadeiras do curso. O trabalho presente serve como método de avaliação para identificar a aplicação dos conceitos teóricos estudados na disciplina de Redes de Computadores e os conceitos práticos estudados na disciplina de Laboratório de Redes de Computadores.

A seguir será descrito qual foi a problemática que motiva o desenvolvimento dessa aplicação, junto veremos como foi a implementação do mesmo, dificuldades, facilidades e escolhas realizadas.

1.2. Enunciado

O enunciado do trabalho final apresentava o problema como a implementação de uma aplicação que simule o comportamento de um protocolo de transferência de arquivos orientado a conexão utilizando apenas o protocolo *UDP*.

Estabelecimento e encerramento de conexão, sequenciamento de mensagens com controle de erro, envio de dados e controle de congestionamento deveriam ser implementados para garantir o funcionamento da aplicação.

[1]guilherme.rizzotto@edu.pucrs.br
[2]joao.leao@edu.pucrs.br

Além disso, a aplicação deveria utilizar duas técnicas de controle de congestionamento, *Slow Start* e *Fast Retransmit*, comumente utilizadas pelo protocolo TCP.

1.3. Concepção

O projeto foi imaginado pelo grupo como um sistema que funciona totalmente pelo log da aplicação. O nome do arquivo que deveria ser enviado seria escrito e passado como parâmetro para a aplicação do cliente. Durante o envio de pacotes, todas as tentativas de envios de pacotes seriam notificadas tanto ao cliente quanto ao servidor. Caso ocorresse perda de pacotes no caminho o cliente e o servidor lidariam com o problema transmitindo as informações de qual pacote está faltando.

2. Especificações

2.1. Tecnologias

O sistema foi desenvolvido em Java, uma linguagem de programação orientada a objetos. Na primeira versão do projeto (sem interface) o mesmo foi codificado na aplicação *Visual Studio Code*. Adiante no projeto, para criação da interface gráfica foi utilizado o *NetBeans*. O projeto se encontra em um repositório do *GitHub* que utilizamos para controle de versionamento.

No sistema são enviados pacotes através de *sockets* que utilizam O *User Datagram Protocol* (*UDP*) que não possui nenhum tipo de verificação de erros.

2.2. Arquitetura

O modelo de arquitetura utilizado foi cliente-servidor, a é uma estrutura que distribui as tarefas entre o fornecedor de recursos (servidor) e quem solicita o mesmo (cliente).

A ideia do projeto era definir que o cliente tem a capacidade de enviar arquivos para o servidor por meio de segmentação de pacotes e envio de cada uma das partes. Essas partes deveriam ser recebidas pelo servidor, processadas e organizadas da maneira correta.

2.3. Interface

A interface da aplicação foi toda limitada ao terminal onde a mesma é executada. A aplicação do cliente inicialmente solicita que o usuário digite 0 para iniciar um envio e recebe o nome do arquivo que deve ser enviado para o servidor. A interface do servidor solicita que um 0 seja digitado para garantir que a conexão foi estabelecida antes de começar a solicitar os pacotes do cliente.

Depois que todos pacotes são enviados e os arquivos do cliente são recriados na pasta do servidor. ambos apresentam que a conexão foi encerrada e que o arquivo foi salvo/enviado.

3. Implementação

3.1. Primeiros Passos

Os primeiros passos para a implementação do trabalho foram simples, a criação de um objetos e classes que seriam utilizadas tanto pelo servidor quanto pelo cliente foram feitas nessa etapa.

3.1.1. *Variables.java*

Essa classe tem uma funcionalidade muito básica, definir algumas variáveis que poderão ser usadas em qualquer classe da aplicação. Essas variáveis são em grande parte cores que foram definidas apenas uma vez e reutilizadas para que sejam utilizadas na interface do terminal e tornar mais intuitivo e compreensível o que acontece no programa. O formato dessas variáveis é o seguinte.

```
1  public static String black = "\u001B[30m"; //definição do preto
```

3.1.2. *PacketObject.java*

A classe *PacketObject* foi responsável por criar o objeto de mesmo nome, que foi essencial para garantir a organização do projeto e manter uma consistência no código. O mesmo é composto das variáveis que são enviadas em um pacote (representação de um segmento do arquivo com seus devidos dados).

Número do pacote, número total de pacotes, nome do arquivo de texto original, conteúdo do arquivo fragmentado, tamanho total do pacote e cálculo de verificação da integridade dos dados (*CRC*) são algumas das variáveis que compõem o objeto.

Além dos *getters*, *setters* e método de construção do objeto, o mesmo é responsável por fazer o cálculo do *CRC* e por retornar um pacote (que será enviado para o servidor ou recebido de um cliente). Esse pacote possui no seguinte formato.

```
1  00                    //número do pacote
2  10                    //número total de pacotes
3  file.txt              //nome do arquivo que foi retirado
4  5584125845            //número CRC
5  arquivo de teste      //conteúdo do pacote
```

Esses pacotes devem possuir um tamanho de 512 *bytes*, mas a aplicação permite que esse valor seja alterado para realização de testes na própria classe principal do cliente e do servidor.

3.2. Cliente - Classes e Métodos

3.2.1. *Main.java*

Esta é a classe principal do cliente, a mesma é responsável por criar um objeto ClientConnection, solicitar o início/fim da conexão e solicitar o envio de arquivos.

3.2.2. *ClientFile.java*

A criação dessa classe foi meramente para otimização e organização do código, portanto cada método será explicado individualmente.

O método de abrir um arquivo simplesmente define a variável *content* com todo o texto do arquivo que será enviado.

O método *setPackets* é responsável pela otimização da leitura de dados utilizando mapas de memória. O mesmo cria uma lista de pacotes e preenche-os com o texto do arquivo a ser enviado. Esse método torna o algoritmo mais otimizado pelo fato de evitar que a variável *content* seja lida por inteiro toda vez que um pacote for manipulado.

O método *getSegment* retorna em formato de texto o pacote solicitado (que será enviado ao servidor).

Os outros métodos são simples e servem para formatação e obtenção das variáveis dessa classe.

3.2.3. *ClientConnection.java*

O primeiro passo na execução desta classe, seria estabelecer uma conexão entre cliente e servidor, essa conexão é estabelecida pelo método *open*.

Em seguida seria iniciado o envio de pacotes pelo *sendFile*, que é responsável por enviar todos os dados de arquivo para o servidor, o mesmo foi implementado de maneira que respeite a técnica de *Slow Start* (enviando pacotes exponencialmente).

A cada conjunto de pacotes enviado (definidos pelo *Slow Start*) é recebida uma confirmação do recebimento do mesmo, essa confirmação é processada pelo método *receivePacket* que foi implementado para respeitar a técnica de *Fast Retransmit* (confirmando a chegada dos pacotes e reenviando-os a cada três confirmações erradas).

Depois de todos pacotes serem enviados, o método *finishTransmission* envia o texto "*end*" para que o servidor possa ser encerrado e executa o método *close* para finalizar a transmissão do cliente.

3.3. Servidor - Classes e Métodos

3.3.1. *Main.java*

Esta é a classe principal do servidor, a mesma é responsável por criar um objeto ServerConnection, solicitar o início/fim da conexão e solicitar o recebimento dos arquivos.

3.3.2. *ServerFile.java*

Os métodos presentes nessa classe foram implementados de maneira intuitiva. Desconsiderando os *getters*, *setters* e *constructor*, temos os seguintes métodos.

O método *addSegment* é responsável por juntar os pacotes recebidos em apenas uma variável para ser mais fácil a inscrição dos mesmos no novo arquivo. Como descrito antes, esse modelo de dados foi criado para otimizar o algoritmo por meio de mapas de memória. Essa otimização faz com que a aplicação não tenha que abrir o arquivo sempre que deseja adicionar dados, mas adiciona-os em uma variável e depois abre o arquivo apenas uma vez para salvá-los.

O método *getPacketObject* transforma um texto em um objeto *packetObject* para ser mais facilmente integrado a outros pacotes e manipulado.

O método *testCRC* verifica se o *CRC* anexado ao pacote é compatível com os dados que foram recebidos.

O método *save* é responsável por salvar o texto dos pacotes recebidos em um arquivo.

3.3.3. *ServerConnection.java*

Como essa classe será executada seguindo uma ordem, as ações e métodos executados estarão apresentados de acordo com a mesma.

Primeiramente é estabelecida uma conexão com o cliente por meio do método *open*.

O método *receiveFile* passa a receber todos os pacotes do cliente e adiciona os dados ao *ServerFile*. O mesmo faz a chamada do método *warn* para avisar o recebimento dos pacotes para o cliente. Caso pacotes que foram recebidos não estejam de acordo com o esperado, o servidor passa a avisar o cliente que eles foram perdidos, assim corrigindo o erro de acordo com a técnica de *Fast Retransmit*.

Depois que todos os pacotes foram recebidos e adicionados ao *ServerFile*, o servidor espera o recebimento de um pacote com o texto "*end*" pelo método *receiveTransmissionEnd*. para que possa salvar o arquivo recebido e finalizar a transmissão.

4. Fluxo de Operações

Como pode ser observado na imagem abaixo, o cliente eo servidor funcionam de maneira simples, as linhas (coloridas) que deixam a caixa de abstração das estruturas são envios de dados. a linha amarela representa um envio que precisa de confirmação. Já as linhas verdes representam dados que não precisam de confirmação (onde na maioria dos casos agem como tal).

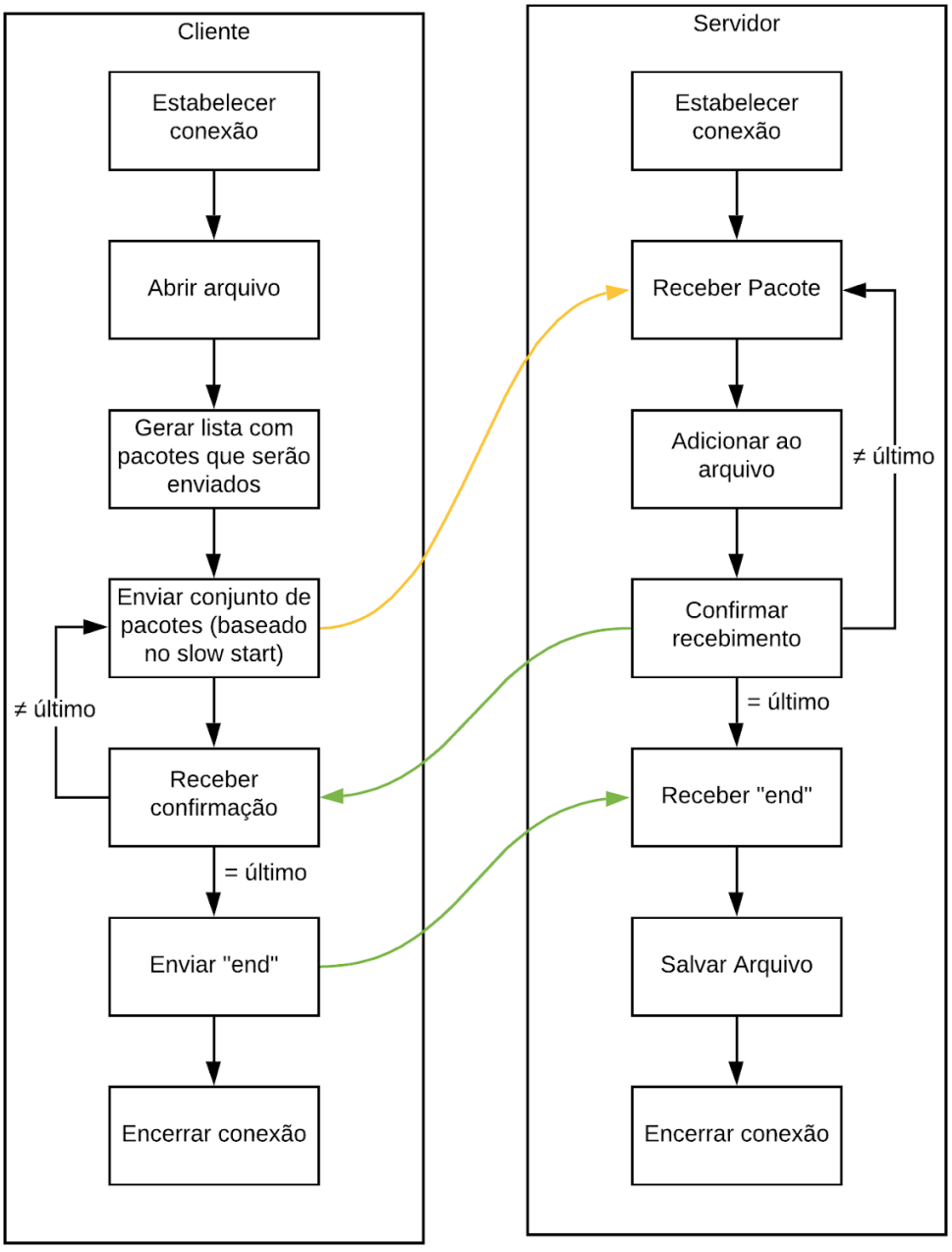


Figura 1: Diagrama de funcionamento do código

5. Fluxo de Dados

Para melhor visualização dos resultados, o grupo optou pela criação de um exemplo visual que mostra o envio de pacotes do cliente para o servidor.

5.1. Sem perda ou atraso de pacotes

Na imagem abaixo, podemos ver o melhor caso possível, sem nenhuma perda ou atraso de pacotes. Neste caso, o arquivo foi dividido em sete partes.

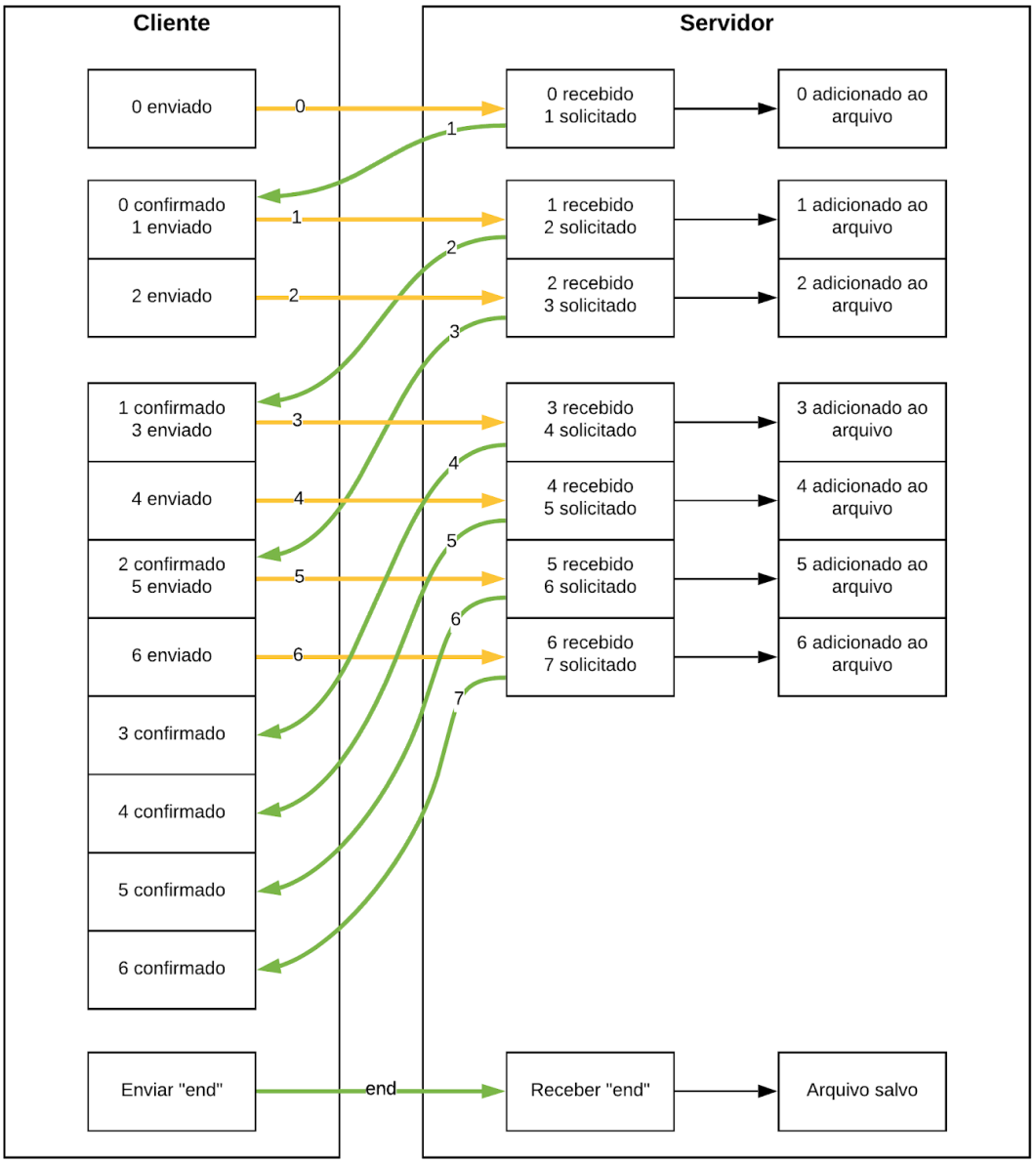


Figura 3: Fluxo de dados sem perda ou atraso de pacotes.

5.2. Com perda ou atraso de pacotes

Considerando que a perda de pacotes pode ocorrer, tanto do cliente para o servidor (envio) quanto do servidor para o cliente (confirmação). Serão apresentados os dois exemplos abaixo. Junto será apresentada a resposta da aplicação para reconstruir o arquivo da maneira correta nos servidor. Nestes casos foi usado um arquivo que foi dividido em 3 partes.

5.2.1. Perda no envio

Neste primeiro caso, a perda ou atraso de pacotes ocorreu no envio do mesmo. Para corrigir isso, o servidor está programado para continuar solicitando o próximo pacote (*ack*). Quando o cliente se torna ciente de que o servidor não recebeu algum pacote, ele passa a enviar o pacote solicitado pelo servidor, corrigindo esse erro.

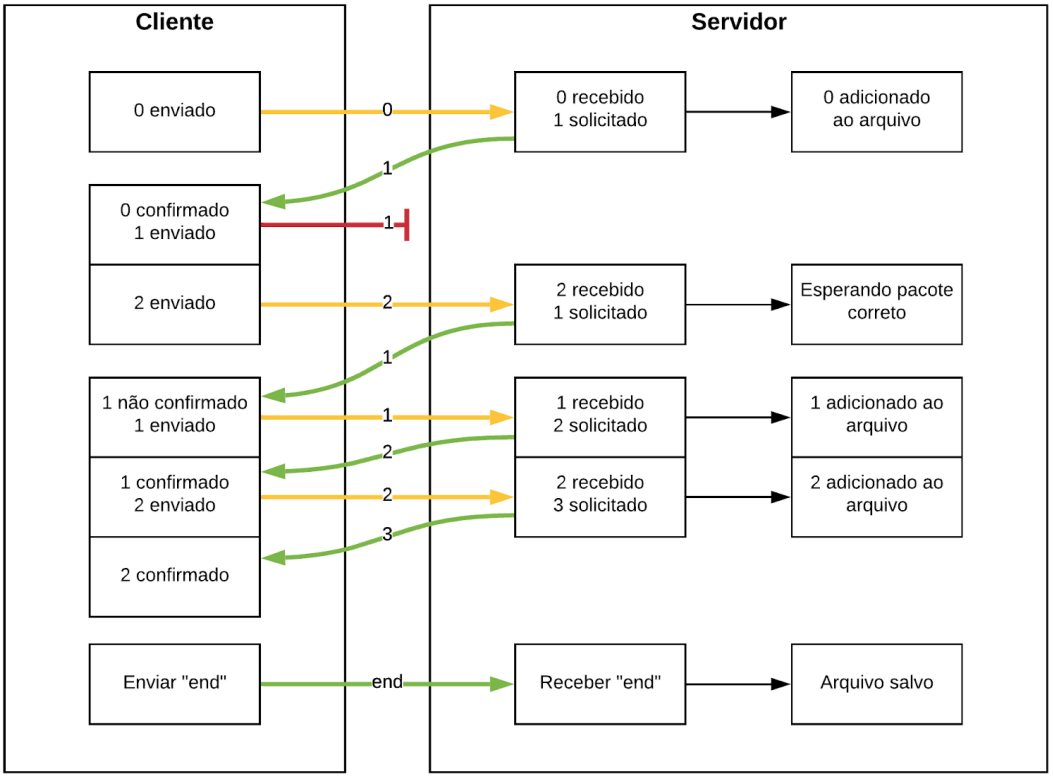


Figura 3: Fluxo de dados com perda de pacotes no envio.

5.2.3. Perda na confirmação

Neste segundo caso, a perda ou atraso de pacotes ocorreu na confirmação dele. Como o único contato do cliente com o servidor é por meio desses pacotes, o cliente não pode constatar onde ocorreu a perda, portanto ele parte do pressuposto de que a perda foi no envio e faz o envio do pacote novamente.

Quando o pacote chega no servidor, é observado que o mesmo já foi adicionado ao pacote, portanto o mesmo é descartado e é solicitado o próximo pacote que deverá ser adicionado ao arquivo.

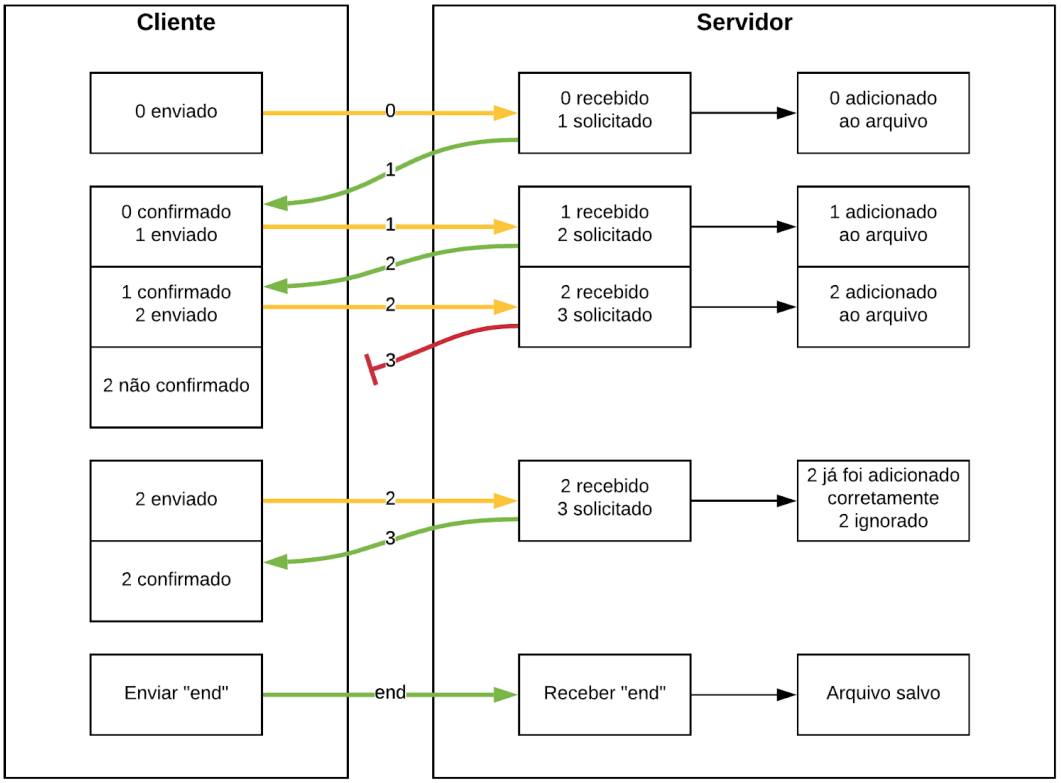
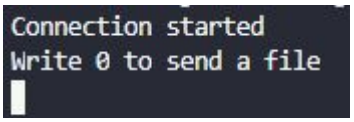
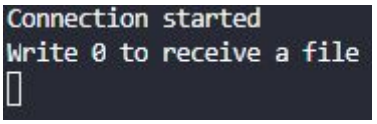
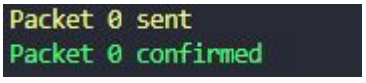
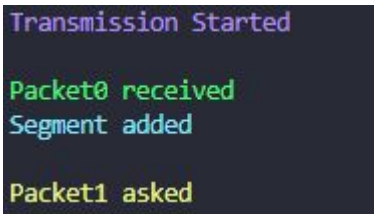
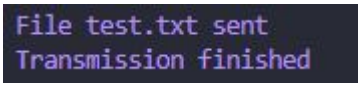
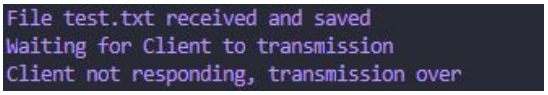


Figura 3: Fluxo de dados com perda de pacotes na confirmação.

7. Interface gráfica

Abaixo encontram-se os resultados obtidos pela aplicação depois de executada no melhor dos casos, ou seja, sem a perda ou atraso de pacotes. As duas colunas apresentam linhas do tempo que acontecem de maneira quase simultânea.

Representação da execução do programa

Cliente	Servidor
Tela inicial do Cliente 	Tela inicial do Servidor 
Solicitação de nome do arquivo a ser enviado 	Esperando ínicio de transmissão do cliente 
Pacote 0 enviado (em amarelo) e confirmação de que o pacote foi solicitado (por meio do ack que apresenta um número superior ao enviado) 	Início de transmissão, recebimento de pacote, adição do segmento a lista de textos recebidos e solicitação de novo pacote (ack). 
Depois de todos pacotes serem enviados, finalizar transmissão enviando um pacote contendo “end”. 	Depois que todos pacotes foram recebidos e que o cliente encerrado finalizar a conexão. 

8. Testes na Interface Gráfica

Como grande parte da execução do trabalho consiste em prevenir erros nas transmissões, essa seção do relatório será dedicada a apresentar alguns possíveis erros e como o programa contornar essas situações para entregar os resultados correto.

Nesse primeiro caso podemos observar que o slow start está funcionando de maneira correta, a medida que as informações são recebidas, novos pacotes são enviados tornando o envio exponencial. Neste caso, o pacote de número 8 não foi confirmado como recebido pelo servidor. O problema pode ter ocorrido no envio ou na confirmação, mas independente disso, o reenvio dele e o recomeço do slow start garante que o mesmo chegará ao destino.

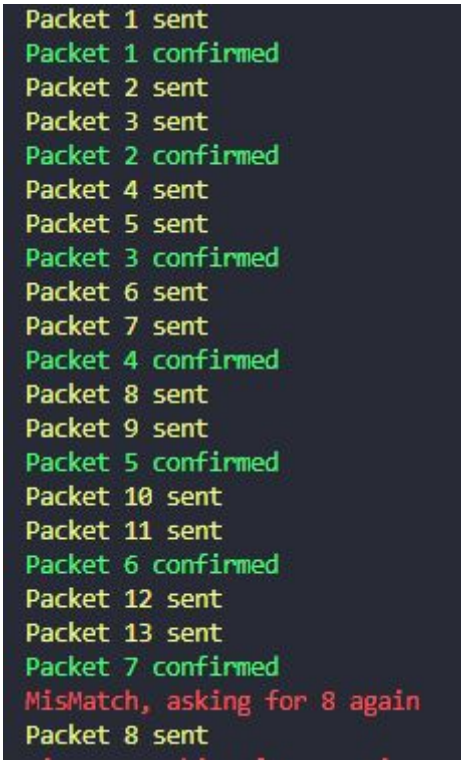


Figura 3: Falha no envio de pacotes e reinício do slow start.

No caso abaixo, podemos comprovar uma dúvida gerada acima. O pacote 8 depois de enviado pelo cliente foi recebido com sucesso pelo servidor, na hora de confirmar o recebimento desse pacote (ack), a confirmação foi perdida, deixando assim o servidor parado até que os próximos pacotes corretos voltem a ser recebidos.

Depois da ocorrência do MisMacth apresentado na imagem acima, o cliente voltou a enviar o pacote 8 que foi recebido pelo servidor, porém não adicionado. O servidor decidiu não adicionar o pacote pelo fato do mesmo já ter sido recebido de maneira correta anteriormente, apresentando a mensagem “Segment already added” que pode ser vista na imagem abaixo.

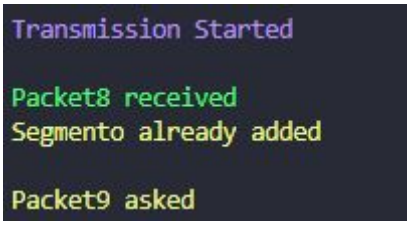


Figura 4: Resposta do Servidor em relação ao recebimento de pacotes repetidos.

O caso abaixo mostra um erro simples. A falta de conexão do servidor (seja pelo fato do usuário não tê-lo iniciado ou pelo fato do mesmo ter sido encerrado de maneira precoce) apresentará o seguinte erro no cliente. Outro caso em que este erro pode ser verificado, é quando pacotes foram perdidos no envio, fazendo com que o servidor não possa confirmar o recebimento assim fazendo o cliente achar que o houve problemas no pacote.

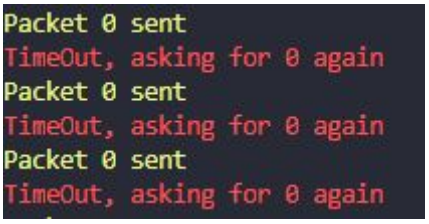


Figura 5: Resposta do cliente quando o envio não recebe uma confirmação de recebimento (ack)

Outro caso simples de erro seria depois de iniciar o servidor. Após 10 segundos de ser digitado o 0 para estabelecer uma conexão e começar a receber um arquivo, caso nenhum pacote seja recebido, o servidor para de receber arquivos e espera que o usuário tente iniciar a conexão novamente.

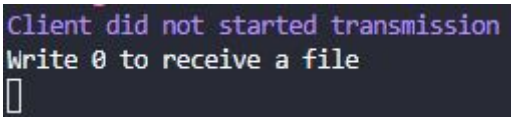


Figura 6: Resposta do servidor quando não é iniciado o envio de um arquivo em 10 segundos.

9. Como Executar o Projeto

Como o projeto funciona por completo no terminal, não foi possível criar um arquivo executável que fosse compatível essa interface. O programa ainda pode ser executado da seguinte forma. Primeiro o usuário deve entrar na pasta build do ClientProject e rodar os seguintes comandos.

```
1  java Main
```

Isso irá iniciar a execução do cliente no terminal que foi selecionado (para melhor visualização da execução, execute o programa em um terminal com suporte para utilização de cores, caso contrário, a leitura dos dados ficará mais poluída)

Para executar o servidor basta digitar os comandos abaixo na pasta build do ServerProject (tudo isso em um terminal diferente do cliente).

```
1  java Main
```

A partir daqui é possível seguir as instruções apresentadas na aplicação e visualizar os resultados no final.

10. Conclusão

Esse trabalhou revisitou os conceitos apresentados nos trabalhos anteriores da disciplina de maneira mais clara. Diversas vezes na implementação o grupo se deparou com situações em que vimos que conhecimentos estabelecidos agora (no final da disciplina) poderiam ter ajudado imensamente nos outros trabalhos. Graças a essa trajetória de aprendizado e método de ensino estudo baseado na CBL (Challenge Based Learning) o grupo conseguiu consolidar mais ainda os conhecimento tanto da disciplina de Laboratório de Redes de COmputadores quanto da disciplina de Redes de computadores.

Referências

Lipsum generator: Lorem Ipsum - All the facts - disponivel em <https://www.lipsum.com/&gt/> último acesso 26/06/2020

Lucidchart - App Lucidchart - disponivel em <https://app.lucidchart.com/\> último acesso 26/06/2020

Wireshark - Stable Release (3.2.2) February 26, 2020 - disponível em <https://www.wireshark.org/> último acesso 26/06/2020

Visual Studio Code - User Setup (1.45.1) May 14, 2020 - disponível em <https://code.visualstudio.com/> último acesso 26/06/2020

clumsy (0.2) - disponível em <https://jagt.github.io/clumsy/> último acesso 26/06/2020

WinMD5Free (1.20) - disponível em <https://www.winmd5.com/> último acesso 26/06/2020