ICEA - UFOP
Redes de Computadores II
Relatório do Trabalho Prático

Icaro Bicalho Quintão[1], **Rafael Fernandes**[1]

icaro.quintao@aluno.ufop.edu.br, rafael.almeida@aluno.ufop.edu.br

[1]Instituto de Ciências Exatas Aplicadas – Universidade Federal de Ouro Preto (UFOP)
Rua 36, 115 Bairro Loanda – João Monlevade, MG – Brasil – 35931-008

1. Introdução

O objetivo maior deste trabalho prático é implementar um serviço de transmissão confiável, utilizando o UDP como protocolo de transporte, sendo o desafio desta aplicação nestes parâmetros a construção de tal confibilidade ao transportar os dados pela rede.

Em sua contrução optamos por utilizar uma abordagem de fácil entendimento e utilização simples : " Stop-and-wait " . As tentativas de utilizar Janela Deslizante foram falhas pois tivemos dois impecilhos distindos relacionados a arquivos grandes transmitidos de acordo com linguagem de programação usada (JAVA), sendo estes :

- Foi criado uma lista de confirmação em formato de threds que aguardavam a resposta por parte do cliente, o alto número destas threds estouravam a pilha de execução da linguagem, finalizando o programa;
- O arquivo a ser transmitido permanecia aberto três vezes (janela deslizante, buffer e cliente) sendo assim estourava a memória disponibilizada pela linguagem, finalizando o programa;

A abordagem utilizada foi essencial para o funcionamento em perfeito estado do sistema proposto com resalvas de algumas dificuldades de implementação expostas nos próximos capítulos aseguir deste artigo.

2. Desenvolvimento

O trabalho desenvolvido consiste em implementar a comunicação entre cliente e servidor utilizando como protocolo da camada de transporte o UDP. Como o UDP não faz tratamentos de erros de diversas naturezas, cabia a nós tratarmos esses erros. Essa comunicação deveria ser implementada em um Servidor de Arquivos, no qual um cliente cadastrado poderia baixar arquivos presentes nesse servidor. A linguagem de programação escolhida para o desenvolvimento foi o Java, que é uma das linguagens que mais possuem conteúdo de camada de transporte na internet, visando eventuais dúvidas da dupla e também por ser uma linguagem que possui portabilidade, que é interessante para eventuais testes em máquinas diferentes e até mesmos Sistemas Operacionais diferentes.

2.1. Classes

A classe Packet foi a primeira a ser implementada, ela possui um número, um vetor de flags e uma String como Body. Como o DatagramSocket do Java só envia bytes, era necessária a abstração de um pacote a fim de se conseguir transferir informações. O método toString() contido nessa classe codifica todas as informações da classe e essa String pode ser enviada em forma de bytes e depois decodificada no cliente para recriar o pacote.

Não houveram problemas com o desenvolvimento dessa classe, um fórum que utilizamos para nortear o trabalho e em especial essa classe foi o Stack Overflow [Overflow 2020]. Um ponto importante aqui é que no momento da codificação, as variáveis de instância são divididas por chaves "}", o que faz com que a nossa aplicação não consiga transferir arquivos .txt que contenham esse tipo de caractere.

O primeiro passo para a comunicação dos pacotes foi desenvolvido de maneira bem simples entre cliente e servidor, da seguinte forma: entre um pacote e outro foi

colocado um timer de 50 milesegundos, sem ACK, ou seja, sem confirmação por parte do receptor. Ao retirar esse timer, no entanto, observou-se que o receptor UDP do Java era muito mais lento que o emissor, fazendo com que menos de 30% dos pacotes enviados chegassem ao destino. Para resolver esse primeiro obstáculo diminuímos o tamanho do tempo do timer, dessa forma tínhamos o ambiente perfeito pra implementação do tratamento dos erros: alguns pacotes se perdiam e alguns chegavam fora de ordem.

Como o envio de pacotes não poderia parar para enviar ACKs, o envio de pacotes é feito em na porta 1996, como solicitado nas diretrizes do projeto e o envio de ACKs em outra, a porta 3000.

2.2. Janela Deslizante

No receptor, a implementação era simples, somente enviar (e reenviar caso necessário) o ACK do último pacote recebido na ordem correta. Isso não mudou durante as tentativas de implementar a Janela Deslizante. Na classe que fazia o envio, entretanto, encontramos alguns problemas.

Em um primeiro momento, cada pacote enviado dava inicio a uma thread cujo objetivo era receber o ACK daquele pacote. Para isso, havia uma lista sincronizada que, caso o ACK chegasse, a thread removia o pacote confirmado da lista e finalizava sua operação. Caso o ACK não chegasse e houvesse timeout, a thread pegava o pacote da lista sincronizada e colocava em outra lista, também sincronizada, a de reenvio. Na classe principal, um pacote era enviado da janela e colocado na primeira lista sincronizada para aguardar pelo ACK. A cada ciclo, a janela andava caso nenhum pacote presente na lista de reenvio fosse o primeiro da lista janela. Essa abordagem funcionou em arquivos pequenos, porém, se fosse aumentado o tamanho do arquivo rapidamente essas threads que aguardam os ACKs estouravam a pilha do Java e o programa emissor finalizava. Ou seja, não era viável para nosso projeto que tem como objetivo a transferência de +-1MB.

Como na primeira tentativa o problema era o número de threads, tentamos então criar uma única thread, responsável por receber todos os ACKs. Agora, tínhamos 3 listas sincronizadas, a janela, o buffer e a de reenvio. O buffer servia para representar os pacotes da janela que já foram enviados, manter as duas era a solução para captar ACKs duplicados. O primeiro ACK chega e é procurado na janela, se estiver lá, é removido, se não estiver, significa que esse ACK é duplicado, então pegava-se uma cópia do pacote seguinte no buffer e jogava pro reenvio. A lógica de reenvio era a mesma da tentativa anterior, com a diferença que, uma vez reenviado, o pacote voltava pro buffer, pro caso de ocorrer outro problema no envio. O problema aqui foi novamente o tamanho do arquivo, uma vez que o arquivo acabava aberto 3 vezes (na janela, no buffer e no receptor), havia estouro da memória do Java.

Foram feitas modificações para tentar diminuir o uso de memória no receptor, como por exemplo salvar o arquivo como append (linha por linha), o que aumentou consideravelmente a complexidade do receptor, uma vez que haviam muitos pacotes fora de ordem. No emissor também tentamos implementar a mesma ideia, ler linha a linha ao invés de ler o arquivo inteiro. O problema era que o pacote demorava muito pra sair do buffer, o que mantinha uma grande parte do arquivo na memória ainda.

Cada uma das tentativas comentadas acima foram feitas mais de uma vez, a fim de tentar resolver os erros que elas causavam, entretanto pela falta de tempo e por estarmos

em um período atípico de PLE, foi decidido pela dupla, a implementação da abordagem stop-and-wait [Papa 2007].

2.3. Stop-and-wait

Para que a complexidade do tratamento de erros do stop-and-wait fosse feita inteira no mesmo lugar, foi criada a classe UDPSocket. A classe possui 2 parâmetros, o primeiro diz se o socket é de envio ou de recebimento e o segundo é se ele é para enviar um ACK ou não. Para o uso exterior dessa classe, basta instanciá-la e utilizar os métodos send(Pacote) ou recieve() que retorna um pacote. Dentro dela, no entanto, os métodos send, recieve, sendAck e recieveAck possuem uma simulação de pacotes perdidos e um tratamento para tal. Um detalhe dessa implementação que foi um pouco problemática é que não podem estar instanciados dois UDPSocket de recebimento ao mesmo tempo. Uma vez que eles utilizam a mesma porta, logo, sempre que utilizado um UDPSocket ele deve ser imediatamente fechado, ainda mais se Cliente e Servidor estão na mesma máquina, que é o caso dos testes que vamos mostrar nesse relatório.

A aplicação em sí era simples de implementar, o único problema foi o sincronismo, uma vez que, como dito anteriormente, não pode ter dois sockets ouvindo a mesma porta ao mesmo tempo. A princípio a tentativa foi sincronizar manualmente, tentar garantir no código do cliente e do servidor que esses sockets não atrapalhassem uns aos outros. Isso falhou devido a quantidade de pacotes que acabaria sendo trocada entre emissor e receptor.

O que funcionou foi adicionar um bloco try-catch dentro de um while no construtor do socket, dessa forma, o erro que dava quando abriam-se dois sockets na mesma porta era burlado e o programa se sincronizava sozinho. Essa solução foi inspirada em um projeto do GitHub [Skoczek 2017].

3. Resultados

A aplicação funciona da seguinte forma:

- O servidor é iniciado e aguarda pela conexão (Figura 1).
- É enviado ao cliente, quando ele é inicializado, as opções de entrar com um usuário e senha já existentes ou cadastrar um novo usuário (Figura 4).
- Uma vez logado, é exibido um menu com 2 opções: listar os arquivos presentes no servidor para que seja efetuado o download por parte do cliente ou finalizar a execução (Figura 5).
- Se escolhida a primeira opção, o servidor manda pro cliente uma lista de arquivos que o cliente pode baixar juntamente com a opção voltar ao menu anterior (Figura 5).
- Se for escolhida a segunda opção, o cliente e o servidor finalizam a execução (Figura 7).

São tratados erros de entradas de nome de usuário ou senha inválidas.

4. Avaliação

Tamanho do arquivo md5.txt transferido: 44KB ou 45056 bytes, tempo que demorou para ser transferido: em torno de 32 segundos, portanto nossa vazão foi de 1408 B/s. Durante a

transferência, foram enviados 494 pacotes, desses 108 pacotes tiveram que ser reenviados pelo não recebimento do ACK, ou seja, a cada 5 pacotes, 1 precisava ser reenviado, o que explica a baixa vasão apresentada pelo nosso sistema. Uma comparação interessante de se fazer aqui é com a vasão do TCP calculado na prática 4, lá calculamos 169042 B/s. Pelo fato de utilizarmos testes em uma unica máquina, o nosso RTT é quase inxistente, através do wireshark podemos chegar ao valor de 0,9 milisegundos de tempo para a transferência de um pacote para outro. Largura de Banda (BW): (tamanho de um pacote) 83 bytes $\div 0,00097 seg(tempodetransmissão) = 85 KB/s$

Já o arquivo Alice.txt, comentado nesse relatório e utilizado em uma das práticas do Wireshark, possui 1052672 bytes e demorava em torno de 1020 segundos para ser transferido, ou seja, nossa vazão cai para algo em torno de 1032 B/s.

5. Conclusão

Com uso de stop-and-wait atrelado ao fato de cada linha do arquivo corresponder a um pacote, as transferências de arquivo são muito demoradas, a transferência de arquivos de +/- 1 MB como pedido no roteiro do trabalho funcionam perfeitamente, porém chegam a demorar em torno de 20 minutos (é o caso do arquivo Alice.txt utilizado durante a prática do Wireshark da disciplina) com a simulação de erros. Como isso foi definido no início de projeto, e não pensamos que esse problema ocorreria, teríamos que alterar o projeto inteiro pra otimizar esse ponto, o que por falta de tempo não é viável.

Novamente por se tratar de um semestre atípico, de poucas semanas, optamos por não desenvolver interface gráfica. Dessa forma, para que se veja todas as transferências e erros, é necessário que abra o arquivo de Log.txt (Figura 10) ao finalizar a aplicação. Há esse arquivo tanto no cliente quanto no servidor, ele possui um timestamp seguido do acontecimento (pacote enviado/recebido ou algum erro, assim como os ACKs) em cada linha.

Um ponto importante que vale ressaltar aqui é que nosso sistema consegue transferir outros tipos de arquivo, não somente .txt, testes com .jpg e .pdf foram feitos, porém dado a alta demora no tempo de transferência em outros tipos de arquivo, optamos por mostrar e focar em arquivos .txt.

O projeto foi desenvolvido utilizando o editor de código fonte Netbeans, vale ressaltar aqui que para a execução em outra máquina deve-se verificar os IPs de cliente e servidor, como para efeito de testes utilizamos localhost, o nosso foi o 192.168.1.108. Para rodar basta abrir ambos os projetos no Netbeans, executar primeiro o servidor e em seguida o cliente. Qualquer arquivo listado no arquivo Files e contido na pasta do servidor pode ser enviado ao cliente, essa foi a solução para que os arquivos log, nome de usuário e senha não fossem passados para o cliente.

As Figuras 11 12 e 13 mostra o Wireshark capturando alguns pacotes do nosso sistema. Uma observação interessante é que na Figura 11, aparece o protocolo ICMP que é um protocolo relacionado a erros, mais especificamente ele reporta um erro de porta inalcançável, isso ocorre porque iniciamos o servidor e na porta 1996 que é nossa porta destino e não se tem ninguém escutando até iniciarmos o cliente, como podemos ver na Figura 12, onde o erro não aparece mais.

6. Capturas de Tela

A sequência de capturas de telas a seguir foram realizadas com o sistema rodando em localhost, por pura simplicidade de testes. Testes em múltiplos computadores foram feitos e o sistema funciona perfeitamente.

Figure 1. Servidor Iniciado

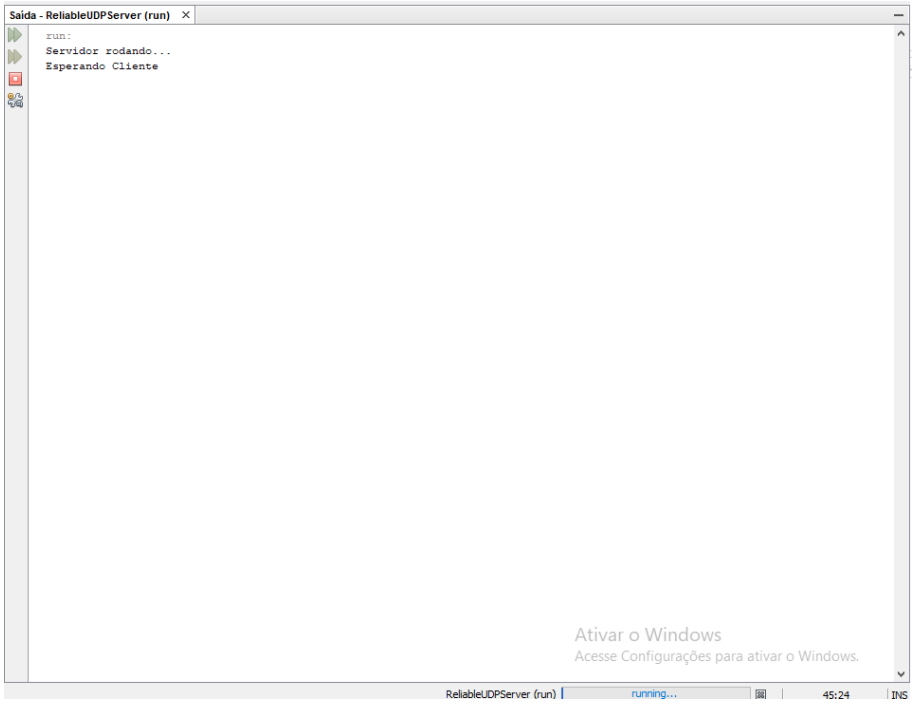


Figure 2. Pasta do Servidor

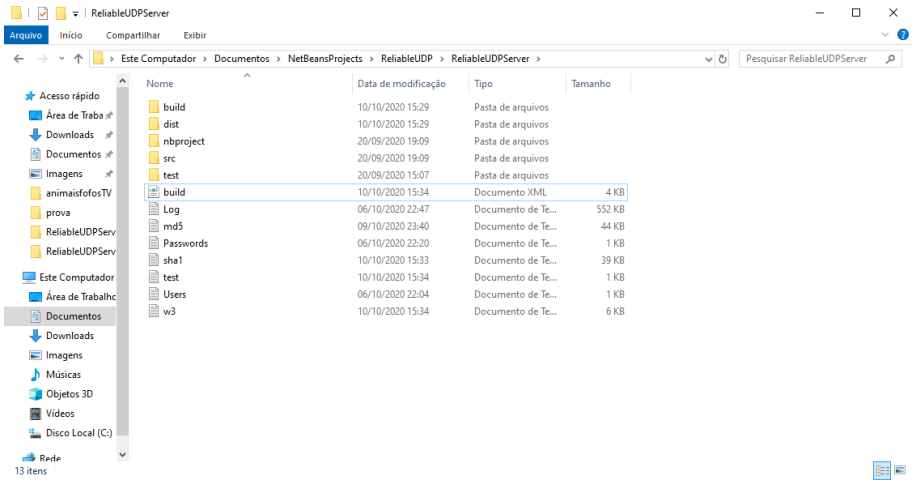


Figure 3. Pasta do Cliente antes de receber o arquivo

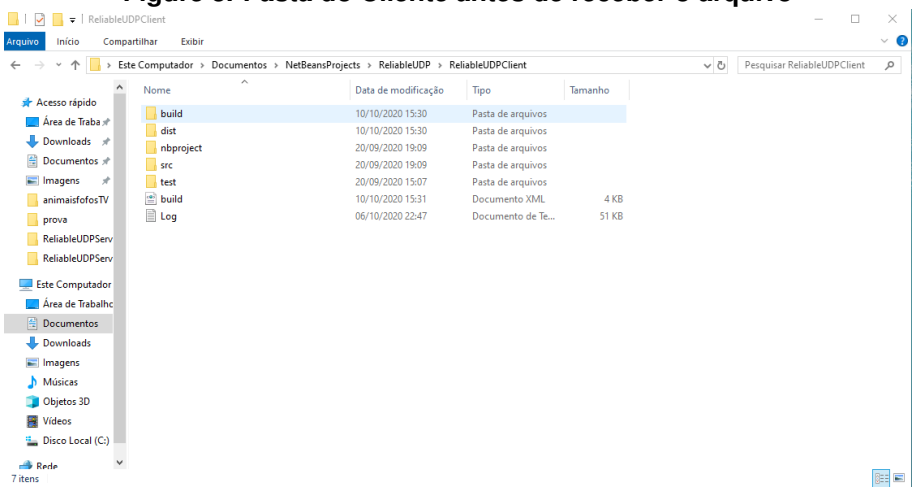


Figure 4. Cliente efetuando Login

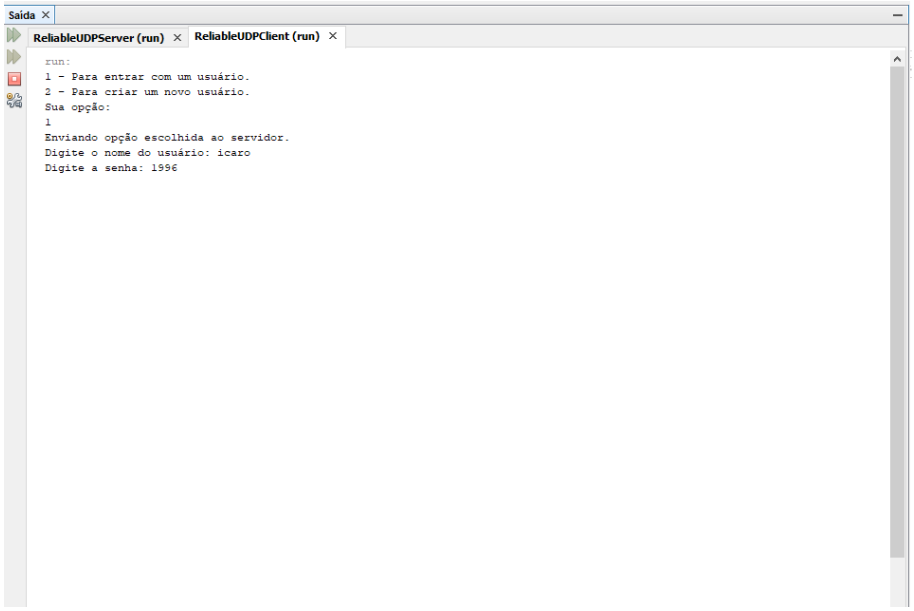


Figure 5. Menu do Sistema

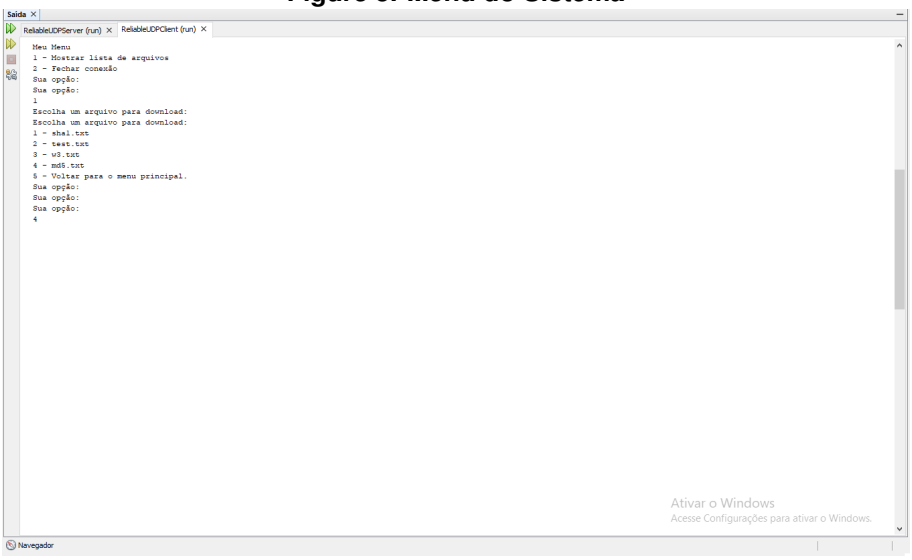


Figure 6. Cliente efetua o download

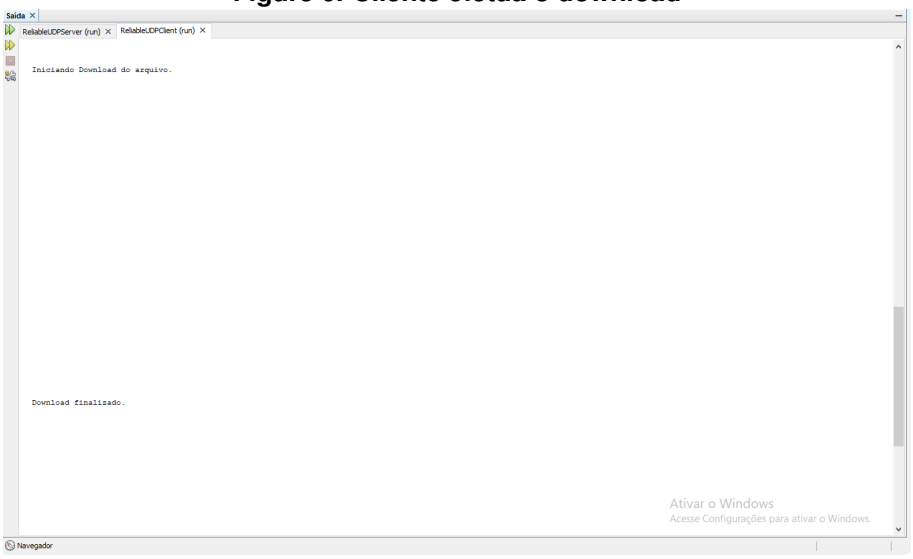


Figure 7. Encerramento do Sistema

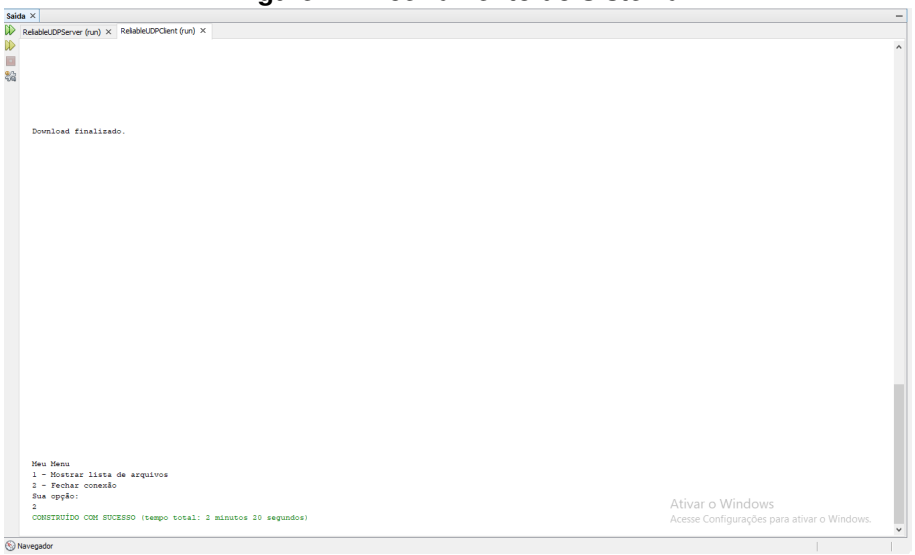


Figure 8. Pasta do Cliente depois de receber o arquivo

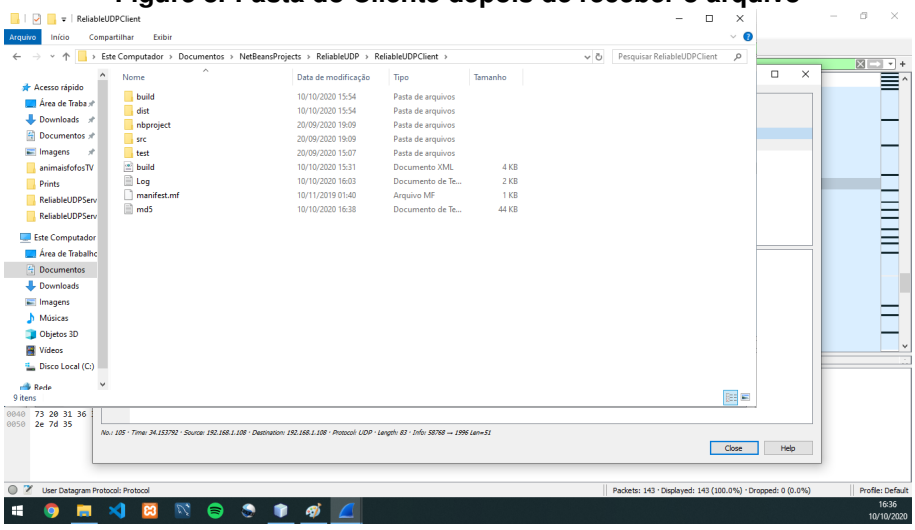


Figure 9. Servidor

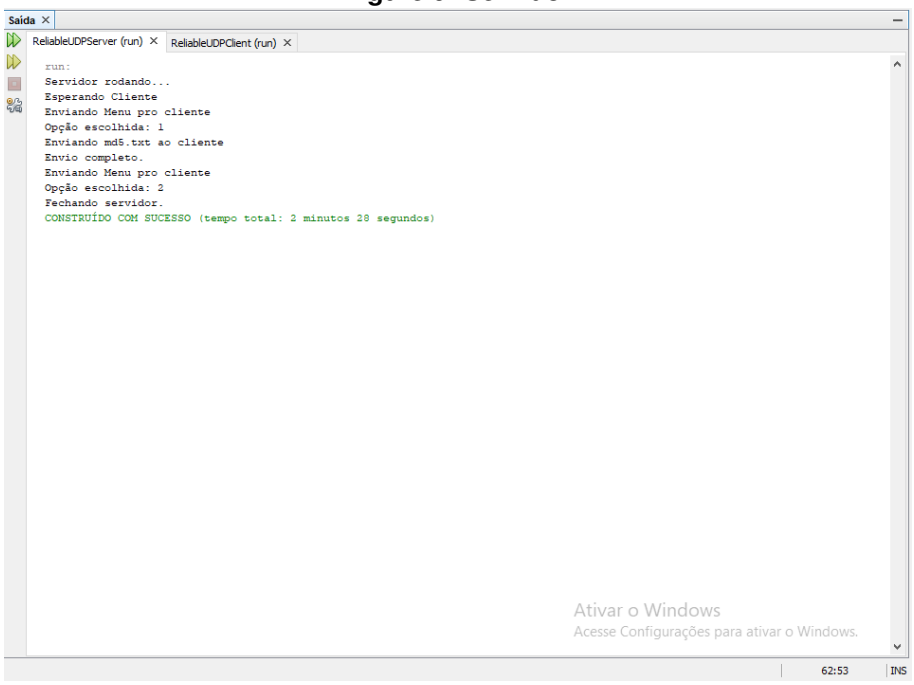


Figure 10. Arquivo Log.txt

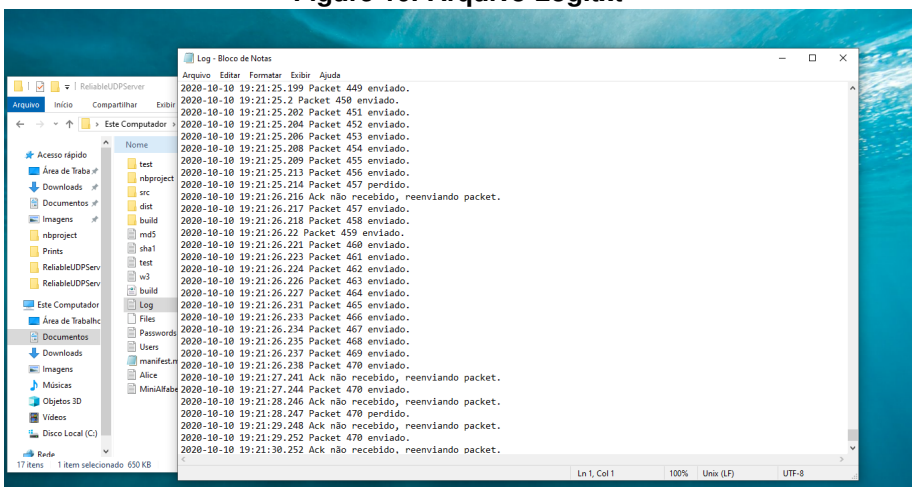


Figure 11. WireShark Erro

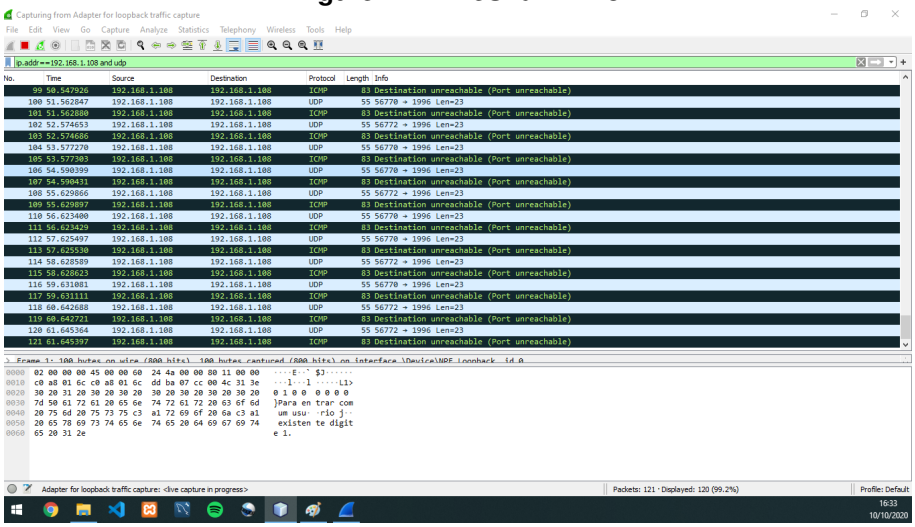


Figure 12. WireShark

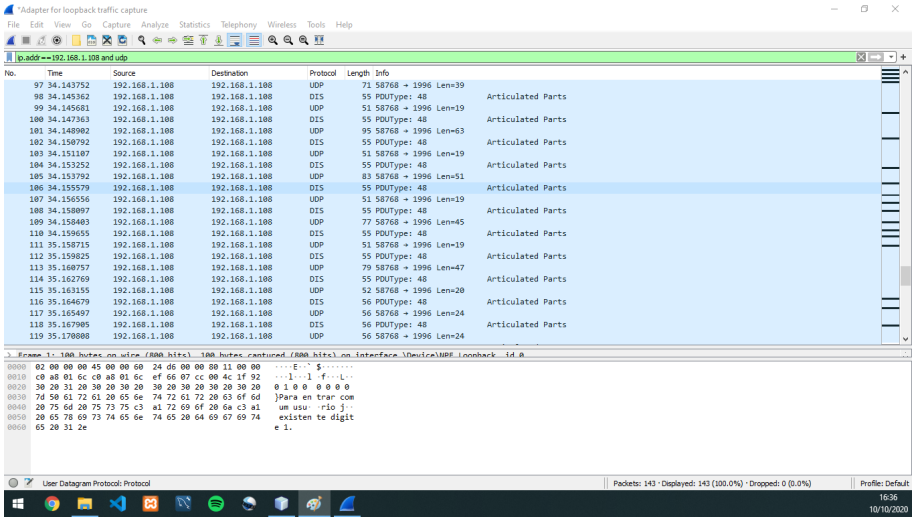
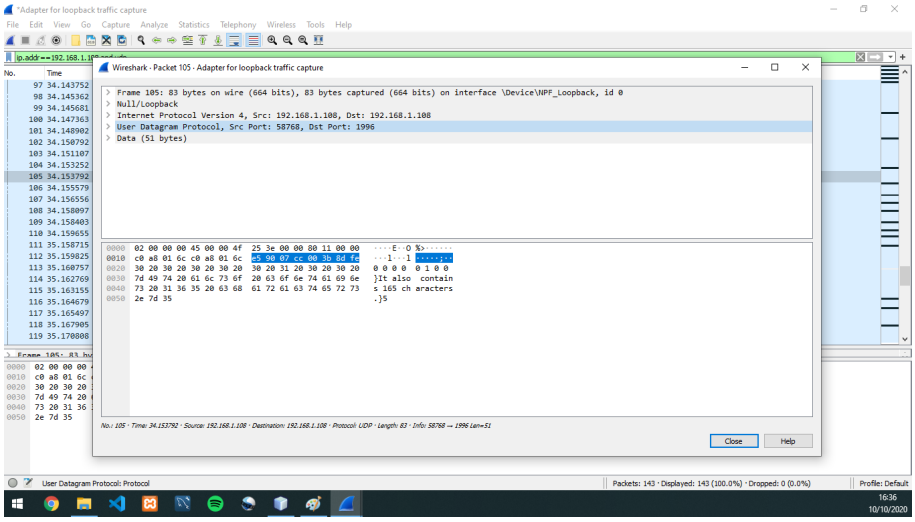


Figure 13. WireShark Detalhes



References

Overflow, S. (2020). Udp packet to class. In *https://stackoverflow.com/questions/46123363/udp-packet-to-class*.

Papa, M. (2007). Project 2: Reliable delivery. In *http://euler.mcs.utulsa.edu/ papama/cs4333/project2.htm*.

Skoczek, L. (2017). Reliable udp project. In *https://github.com/linoskoczek/ReliableUDP*.