T2 Laboratório de Redes

Daniela Amaral, Vinicius Lima

Escola Politécnica - PUCRS

Porto Alegre – RS – Brasil

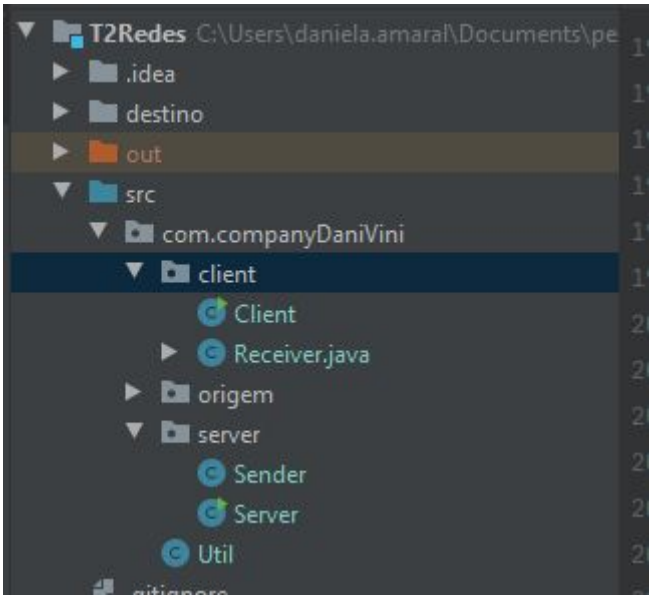
1. Introdução

No escopo da disciplina, o problema principal a ser resolvido pode ser resumido da seguinte maneira: deve-se criar uma aplicação que utilize um protocolo de conexão UDP que simule o comportamento de um protocolo de transferência de arquivos orientado à conexão. A aplicação deverá implementar duas técnicas de controle de congestionamento utilizadas pelo protocolo TCP: Slow Start e Fast Retransmit.

2. Conexão entre servidor e cliente

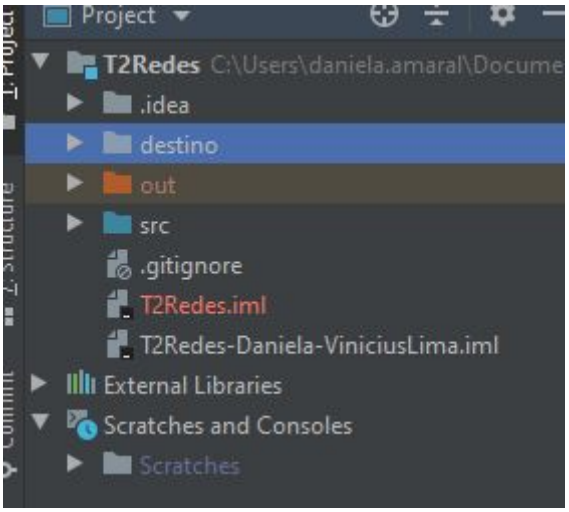
Para resolver o problema inicial, decidimos pela implementação de um algoritmo em linguagem Java para estabelecer a conexão entre computadores. A linguagem Java fornece as classes *Sockets* e *DatagramSocket*, que encapsulam o funcionamento dos métodos de conexão do protocolo UDP, que é um protocolo de comunicação sem garantia de entrega, ao contrário de uma conexão TCP (*Transmission Control Protocol*). Os dois protocolos funcionam sobre o protocolo IP.

A aplicação pode ser dividida em duas partes: o Cliente, composto pelas classes Client e Receiver na pasta client; o Servidor, composto pelas classes Server e Sender na pasta server.



Organização dos pacotes da aplicação

O cliente da aplicação passa uma mensagem para o servidor informando qual arquivo que deseja transferir para a pasta destino. O servidor então envia para o cliente o arquivo, dividido em pacotes de 512 bytes. A conexão é encerrada após todos os pacotes do arquivo serem enviados para o cliente.

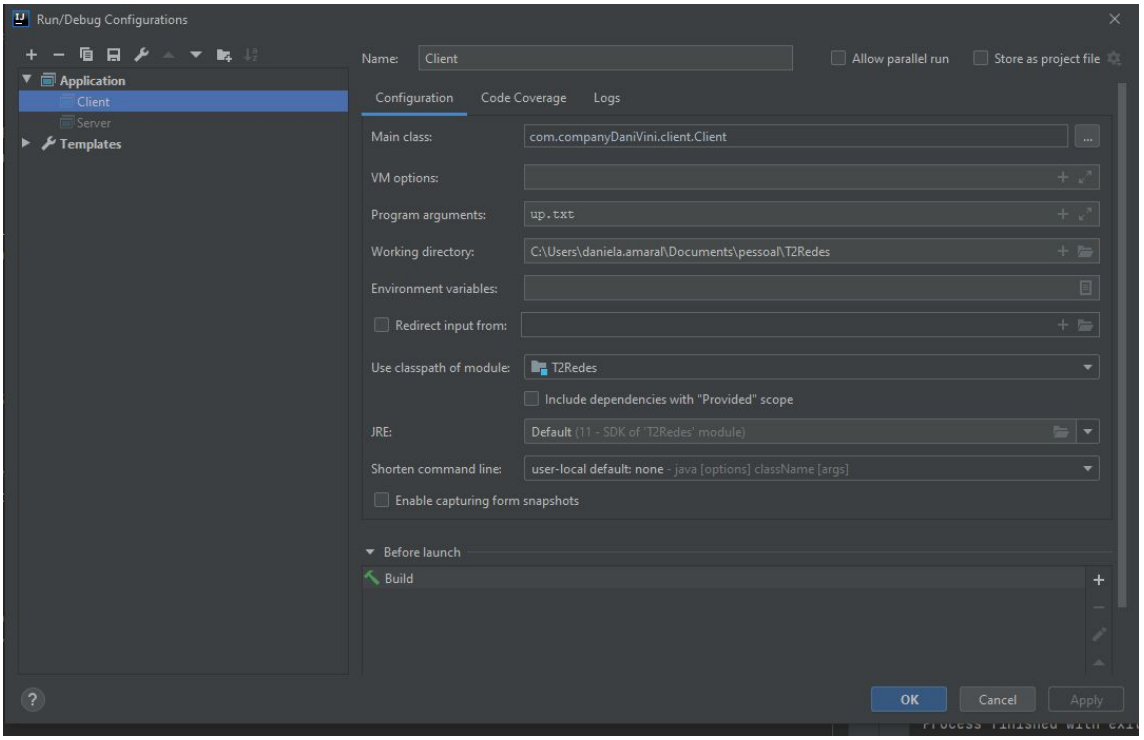


Pasta destino onde os arquivos transferidos são salvos

3. Funcionamento da Aplicação

A aplicação funciona da seguinte forma:

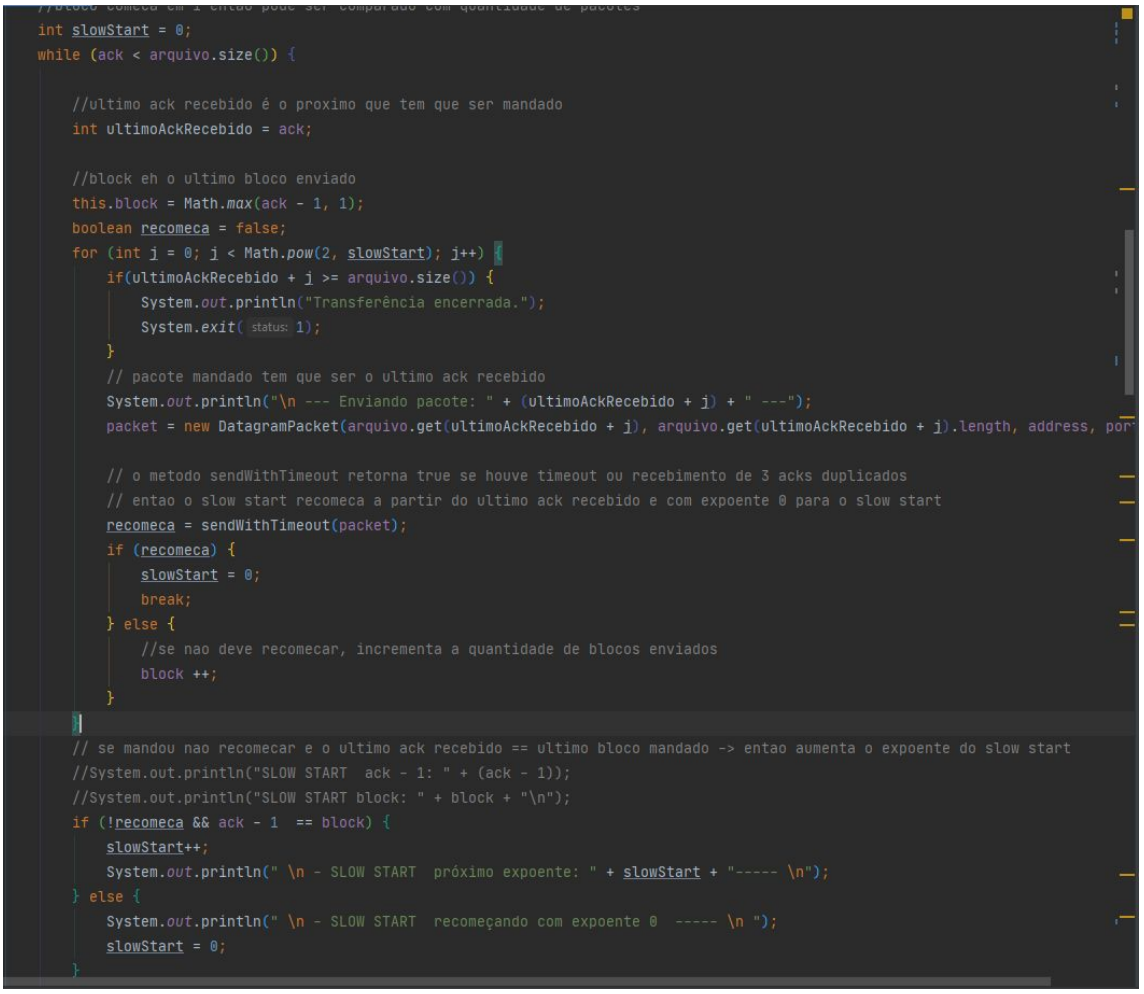
- O Cliente escolhe um arquivo, obrigatoriamente localizado na pasta “origem” dentro do projeto, este arquivo será copiado para a pasta “destino” localizada na raiz do projeto. Se o arquivo já existe na pasta de destino, ele é sobrescrito;
- A escolha deste arquivo é feita de forma que o usuário passa o nome do arquivo como argumento ao rodar o método *main* da classe Cliente;



Exemplo configuração para rodar a classe Main escolhendo o arquivo up.txt

- O servidor recebe o nome do arquivo, e então passa para o cliente uma mensagem com a quantidade de pacotes que serão enviados;
- O servidor le o arquivo que está sendo solicitado, o divide em arrays de bytes de tamanho 512 e os guarda em um ArrayList, para facilitar o acesso posterior para mandar cada pacote, e para se saber antes de iniciar o envio quantos pacotes serão enviados ao total;

- O cliente após receber a quantidade de pacotes, manda uma mensagem ACK;
- O servidor após receber o ACK da quantidade de pacotes, começa a mandar os pacotes para o cliente usando o protocolo Slow Start;



```
int slowStart = 0;
while (ack < arquivo.size()) {

    //ultimo ack recebido é o proximo que tem que ser mandado
    int ultimoAckRecebido = ack;

    //block eh o ultimo bloco enviado
    this.block = Math.max(ack - 1, 1);
    boolean recomeca = false;
    for (int j = 0; j < Math.pow(2, slowStart); j++) {
        if(ultimoAckRecebido + j >= arquivo.size()) {
            System.out.println("Transferência encerrada.");
            System.exit( status: 1);
        }
        // pacote mandado tem que ser o ultimo ack recebido
        System.out.println("\n --- Enviando pacote: " + (ultimoAckRecebido + j) + " ---");
        packet = new DatagramPacket(arquivo.get(ultimoAckRecebido + j), arquivo.get(ultimoAckRecebido + j).length, address, por

        // o metodo sendWithTimeout retorna true se houve timeout ou recebimento de 3 acks duplicados
        // entao o slow start recomeca a partir do ultimo ack recebido e com expoente 0 para o slow start
        recomeca = sendWithTimeout(packet);
        if (recomeca) {
            slowStart = 0;
            break;
        } else {
            //se nao deve recomecar, incrementa a quantidade de blocos enviados
            block ++;
        }
    }
    // se mandou nao recomecar e o ultimo ack recebido == ultimo bloco mandado -> entao aumenta o expoente do slow start
    //System.out.println("SLOW START  ack - 1: " + (ack - 1));
    //System.out.println("SLOW START block: " + block + "\n");
    if (!recomeca && ack - 1  == block) {
        slowStart++;
        System.out.println(" \n - SLOW START  próximo expoente: " + slowStart + "----- \n");
    } else {
        System.out.println(" \n - SLOW START  recomeçando com expoente 0  ----- \n ");
        slowStart = 0;
    }
```

Trecho de código na classe Sender a partir da linha 61 responsável por controlar o envio de pacotes com Slow Start.

- Os pacotes que o servidor manda para o cliente possuem todos 526 bytes, sendo os 2 primeiros bytes com “03”, correspondentes a identificação DATA do TFTP. Os 2 bytes seguintes com o número do bloco que está sendo enviado. Os 10 bytes seguintes com o CRC dos 512 bytes de dados do arquivo que estão sendo mandados no pacote;

- Para cada pacote que o servidor manda ao cliente, é esperado que o cliente mande uma mensagem ACK contendo o número do próximo pacote esperado;
- Nos DatagramSocket utilizados na aplicação, foram estabelecidos timeouts de 500ms tanto para o envio quanto para o recebimento de mensagens no cliente e no servidor, utilizando o método *setSoTimeout()* da classe DatagramSocket;
- O Slow Start foi desenvolvido de forma a serem mandados pacotes de forma exponencial para o cliente, o processo de envio de pacotes consiste em enviar um pacote e receber um ack, não necessariamente correspondente ao último pacote enviado (se houver lentidão ou perda de pacotes). Se forem recebidos 3 ACKS iguais em sequência (controlados por meio das variáveis de classe: *ack* e *ackTriplicado* na classe *Sender*) ou se houver timeout no recebimento de algum ACK, o processo é recomeçado, isto é, é setado o valor da variável *slowStart* da classe *Sender* para 0, a qual controla qual o expoente usado na base 2 para a iteração do envio de pacotes.
- O controle de erros na transmissão do pacote foi feita utilizando cálculo de CRC, feito com as classes *Checksum* e *CRC32* do pacote java.util.zip. Para isto, o CRC foi calculado no envio dos pacotes e inserido junto na mensagem. Ao chegar no cliente, ele calcula novamente o CRC dos dados recebidos e compara com o CRC enviado na mensagem. Se estes não estiverem iguais, a mensagem é descartada.