Redes de computadores EP1

**Nome: Ramon Neres Teixeira Jardim**

**RA: 21202410009**

Link do vídeo:

Link para o repositório: <https://github.com/RamonJardim/Redes-de-computadores-pos/tree/main/EP1>

**Intruções para execução estão no readme.**

# Explicação das funcionalidades

O componente *Channel.java* (canal) foi desenvolvido como uma subclasse da classe *DataSocket* de forma a facilitar o envio/recebimento de mensagens. O construtor na linha 72 do componente faz a leitura do arquivo *config.json* que contém as probabilidades para os possíveis parâmetros das mensagens e armazena os dados na variável *config* (linha 55).

Na linha 88, a função *send* envia segmentos UDP utilizando a função *send* da linha 92. Esta função cria a estrutura do segmento que consiste em 4 bytes representando o checksum (calculado como a soma de todos os bytes da mensagem), 4 bytes representando número de sequência da mensagem (que é incrementado em 1 a cada mensagem) e os demais bytes representando o conteúdo da mensagem. Após a montagem da estrutura, a função checa as probabilidades de cada parâmetro a ser adicionado às mensagens (entre as linhas 116 e 134) e os adiciona conforme o resultado de um gerador de números aleatórios encapsulado na função *randomize* na linha 213. Após aplicar todos os parâmetros aplicáveis, na linha 136 é utilizada a função *send* da superclasse para efetuar o envio da mensagem.

Os parâmetros são aplicados utilizando as funções:

* delayMessage – linha 174
  + Atrasa a mensagem de acordo com o tempo definido no arquivo *config.json*.
* corruptMesage – linha 186
  + Soma 1 em um byte aleatório da mensagem, corrompendo a mensagem.
* cutMessage – Linha 193
  + Corta a mensagem caso seja maior que um dado valor definido no arquivo *config.json*.
* duplicateMessage – Linha 207
  + Utiliza da superclasse para enviar o segmento, que será novamente enviado na função *send* anterior.

Para eliminar a mensagem, a execução da função *send* é finalizada com um *return* na linha 119.

Para o recebimento das mensagens, é utilizada a função *receive* na linha 140, que faz a chamada para a função *receive* da superclasse e divide o segmento nos trechos citados anteriormente [checksum, número de sequência, texto da mensagem].

Os números de sequência das mensagens recebidas são armazenados no *HashMap* *seqNumberMap* da linha 70, cuja chave é a combinação “ip:porta” do remetente e o valor é uma *ArrayList* de inteiros que contém todos os números de sequência recebidos daquela combinação “ip:porta”. Caso seja recebida uma mensagem cujo número de sequência já esteja na lista referente ao remetente, a mensagem é dada como duplicada (linha 163), caso contrário, o número de sequência da mensagem é adicionado à lista e o ACK é enviado (caso a mensagem recebida já não seja um ACK) (linha 166).

Para consolidação dos dados das mensagens enviadas e recebidas, outros *HashMaps* foram criados para armazenar o número de mensagens enviadas/recebidas com cada parâmetro para cada remetente/destinatário. Os *HashMaps sendCount* (linha 59), *eliminateCount* (linha 60), *delayCount* (linha 61), *duplicateCount* (linha 62), *corruptCount* (linha 63) e *cutCount* (linha 64) são responsáveis por, respectivamente, contar as mensagens enviadas, eliminadas, atrasadas, duplicadas, corrompidas e cortadas. Também foram criados os *HashMaps receivedCount* (linha 66), *receivedWithFailedIntegrityCount* (linha 67) e *receiveDuplicateCount (*linha 68) para armazenar, respectivamente, a contagem de mensagens recebidas, recebidas com falha de integridade e recebidas em duplicidade.

Os H*ashMaps* citados são utilizados para consolidar as estatísticas tanto de envio quando para recebimento. Os dados são apresentados de forma apartada por cada conjunto “ip:porta” pela função *consolidateAll* na linha 236. A consolidação de mensagens eliminadas pelo lado recebedor calcula o número de mensagens perdidas contando os números de sequência que não foram recebidos. Por conta disso é possível que o número seja um pouco diferente da realidade, dado que o destinatário não sabe quantas mensagens foram enviadas pelo remetente, por exemplo, caso o remetente envie 4 mensagens [1, 2, 3 e 4] e as mensagens 2 e 4 sejam perdidas, a visão do destinatário seria [1, 3], permitindo uma percepção de que a mensagem 2 foi perdida, mas não tendo ciência da existência da mensagem 4.

# Explicação das *threads*

Existem 2 Threads no código: *ParallelSender* e *ACKListener nas* linhas 10 e 33 do arquivo *Client.java*.

A *thread ParallelSender* é iniciada caso o usuário escolha enviar as mensagens em paralelo para o servidor. Neste cenário, uma *thread* é instanciada e iniciada para cada mensagem a ser enviada (linha 129). Para evitar que o programa termine antes de que todas as mensagens sejam enviadas, na linha 133 é feito o *join* de todas as *threads* iniciadas, de forma que todas tenham concluído o envio ao finalizar a execução. Todos os *HashMaps* no canal são do tipo *ConcurrentHashMap* para evitar condições de corrida com múltiplas *threads* enviando mensagens ao mesmo tempo. Também para evitar condições de corrida, as funções *getSequenceNumber* e *incrementCount* (responsável por incrementar os contadores nos *HashMaps*) possueam a modificador *synchronized* que impede que múltiplas *threads* acessem a função ao mesmo tempo.

Já a *thread ACKListener* é responsável por receber os ACKs vindos do servidor. Uma vez que muitas mensagens podem ser enviadas em sequência, foi necessário desenvolver uma thread para que o cliente pudesse esperar pelos ACKs ao mesmo tempo que seguia enviando mensagens. Esta *thread* é instanciada e iniciada nas linhas 123 e 124 e consiste apenas em um loop infinito aguardando por mensagens com um *timeout* de 1s que quando atingido finaliza a thread. Na linha 142 é feito um join na *thread* e o programa é então finalizado.