Redes de computadores EP2

Nome: Ramon Neres Teixeira Jardim

RA: 21202410009

Link do vídeo: https://youtu.be/faGzlJO1HDY

Link para o repositório: https://github.com/RamonJardim/Redes-de-computadores-pos/tree/main/EP2

Instruções para execução estão no README.md.

Explicação das funcionalidades

O componente *ReliableChannel.java* (canal) foi baseado no componente canal do EP1, agora incluindo retransmissão de mensagens perdidas com o protocolo Go Back N. O construtor na linha 174 do componente faz a leitura do arquivo *config.json* que contém o tempo de timeout, tamanho da janela e as probabilidades para os possíveis parâmetros das mensagens e armazena os dados na variável *config* (linha 149). O construtor também configura o timeout para recebimento de mensagens como o tempo configurado mais 10 segundos (este timeout é utilizado para identificar o fim da transmissão e finalizar a execução do programa).

Na linha 193 a função *send* recebe uma lista de *DatagramPackets* e faz a construção do conteúdo de cada um usando a função *buildSegment* (linha 344), que adicionar o header incluindo *checksum*, número de sequência e uma flag indicando se o pacote é um ACK. Após a construção dos segmentos, cada um deles é armazenado no buffer de envio *sendingDataPackets* (Linha 155). Na sequência, é iniciada a thread que escuta os ACKs e inicia-se o *timer*. Após isso é feito o envio da primeira janela de pacotes usando a função sendWindow (linha 223) e inicia-se um loop para retransmitir a janela em caso de timeout (usando a mesma função *sendWindow*) ou, caso não haja timeout, enviar os pacotes já liberados para envio com a função *sendRange* (linha 232). Após completar os envios com todos os pacotes confirmados pelo Receiver, a thread que escuta os ACKs é interrompida e o código é finalizado. Ambas as funções *sendWindow* e *sendRange* utilizam a função *send* da linha 258 que reaproveita a lógica utilizada no EP1 para adicionar erros às mensagens.

Para o recebimento das mensagens, é utilizada a função *receive* na linha 264, que faz a chamada para a função *receive* da superclasse e divide o segmento nos trechos citados anteriormente [checksum, número de sequência, flag ACK e texto da mensagem]. Esta função também checa se o segmento está corrompido (linhas 277 e 278) e se está

duplicado (linha 289) para então enviar o ACK (caso a mensagem não seja um ACK por si própria). O ACK é enviado nas linhas 292, 298 e 301, utilizando a thread ACKSender, usando o número de sequência da mensagem recebida caso seja a esperada ou o número de sequência da última mensagem recebida com sucesso caso a mensagem recebida esteja fora de ordem ou duplicada. O envio é feito por uma outra thread para que o envio do ACK não impacte no recebimento das mensagens. O envio do ACK passa pelo mesmo processo do envio das mensagens descrito anteriormente.

As mensagens perdidas ou muito atrasadas causarão um timeout no *sender* que irá reenviar a janela de mensagens até que o *receiver* confirme o recebimento. Para o *receiver*, mensagens recebidas fora de ordem ou em duplicata são descartadas e um ACK da última mensagem recebida com sucesso será enviado, o que em algum momento ocasionará um timeout no servidor, que irá reenviar a janela de mensagens. No caso do *receiver*, ACKs duplicados são descartados e ACKs fora de ordem são tratados de forma cumulativa, portanto o ACK 10 confirma que todas as mensagens até o número de sequência 10 foram recebidas pelo *receiver*.

Para consolidação dos dados das mensagens enviadas e recebidas, foi reaproveitada a estrutura do EP1.

Explicação das *threads*

Existem 3 Threads (além da principal) no código: *Timer* (linha 89)*, ACKListener* (linha 61) e *ACKSender* (linha 116).

A *thread Timer* usada para fazer a contagem do tempo para o timeout que causada o reenvio da janela de pacotes. A thread consiste apenas em uma pausa na duração do timeout que quando acaba seta uma flag com o nome timeout (linha 153) para indicar que o tempo foi atingido. Essa flag será lida pelo loop de envio de pacotes que irá fazer a retransmissão. O valor da flag é o número de sequência do pacote que gerou o timeout. Caso o referido pacote seja confirmado antes do timeout ser atingido, a função *stopTimer* (linha 248) será chamada para que a thread seja interrompida.

Já a *thread ACKListener* é responsável por receber os ACKs vindos do receiver. Para não afetar o envio de mensagens, foi criada uma thread para o recebimento dos ACKs. Esta *thread* é instanciada e iniciada nas linhas 200 e 201 e consiste apenas em um loop que consistem em chamar a função receiveACK (linha 326) e que dura até que todas as mensagens sejam confirmadas.

Por fim, a *thread ACKSender* é utilizada para fazer os envios dos ACKs pelo *receiver*. Assim como no caso anterior em que o *sender* precisa receber os ACKs ao mesmo tempo que continua enviando mensagens, o *receiver* precisa enviar os ACKs em paralelo de forma a não interromper o recebimento das mensagens que podem ainda chegar.

Teste remoto e teste local

O teste remoto foi realizado com duas máquinas na AWS, uma no Brasil e outra em Londres. Foi possível observar que no teste remoto, apesar de usar a mesma janela e mesmo timeout, o tempo de execução foi um pouco maior. Isso se deve ao maior RTT (180ms vs 0ms), que multiplicado pelo número de mensagens transmitidas gera um aguardo considerável. A diferença entre os tempos de execução ficou numa média de 20 segundos.

Execuções remotas.

Execuções locais: