DIG-DCC023 - REDES DE COMPUTADORES

TRABALHO PRÁTICO 2

Aluno: Danilo Pimentel de Carvalho Costa
Matrícula 2016058077

Introdução

O Trabalho Prático 2 da disciplina de Redes de Computadores visa aplicar os conceitos aprendidos sobre os protocolos TCP e UDP em um sistema de transmissão de arquivos. Foram dadas diversas opções para escolha de linguagem. A linguagem escolhida para este trabalho foi Python.

Servidor e cliente básicos

Para começar o desenvolvimento do sistema, foram implementadas versões básicas do cliente e do servidor. Nesta versão, o servidor abriria um socket TCP e um socket UDP por cliente. O cliente quando iniciado conectaria no socket TCP primeiramente, e o servidor envia por ele a porta para o socket UDP criado. O cliente então enviaria a primeira linha do arquivo que recebeu como entrada na linha de comando pelo canal UDP. O servidor simplesmente replicaria qualquer mensagem que recebesse de volta para o cliente.

Aqui já foram cobertos alguns requerimentos do trabalho prático. O servidor e o cliente devem suportar ipv4 e ipv6. No caso do servidor, foram criadas duas threads que executam basicamente o mesmo código, porém uma cria um socket ipv4 e outra cria um socket ipv6. No cliente, é feita uma checagem no endereço recebido para determinar se é um endereço ipv4 ou ipv5, e assim decidir qual tipo de socket abrir. Este modelo foi útil para determinar qual tipo de socket udp seria aberto. Se um cliente conecta na thread do servidor que criou o socket ipv6, então o cliente está se comunicando via ipv6. Logo, o socket udp também deve ser ipv6. O raciocínio é análogo para ipv4.

A estratégia para alocação de portas UDP foi a seguinte: é mantido um contador de clientes conectados. Para cada novo cliente, o contador é incrementado em 1. O valor atual do contador somado ao número da porta TCP recebido como entrada resulta na porta UDP para o cliente. O contador é reiniciado quando chega em 1000. Isso impõe um limite de clientes conectados, mas é suficiente para a realização do trabalho. Foi necessária a utilização de locks para evitar condições de corrida na atualização de leitura do valor do contador, já que este é compartilhado para as threads ipv4 e ipv6 do servidor.

Outro requerimento coberto foi o recebimento e validação do nome do arquivo pela linha de comando. O nome foi validado via expressão regular. Uma das dificuldades foi a filtragem de somente caracteres ASCII no nome do arquivo. A solução encontrada foi criar uma expressão regular que realizava checagem no intervalo de caracteres ASCII, excluindo somente o caractere ',' do intervalo, visto que este é permitido somente para definição da extensão.

Canal de Controle

O próximo passo depois de construir o sistema básico foi implementar a lógica do canal de controle. O protocolo do canal de controle estabelece algumas mensagens que são

enviadas para estabelecer conexão, enviar informações do arquivo que será transmitido, etc. A especificação também define o formato e tamanho das mensagens trocadas neste canal.

A primeira tentativa de implementação das mensagens foi utilizando a biblioteca struct do Python. Ela proporciona uma maneira de definir uma estrutura de dados serializável em bytes, lidando com todos os detalhes e sendo bem fácil de usar. O impedimento encontrado no uso desta biblioteca foi o tamanho da mensagem. Pode-se ver que a biblioteca adiciona outros bytes para evitar conflito na formatação da mensagem, gerando mais bytes do que a especificação define. Por isso, o uso desta biblioteca foi abandonado.

Assim, foi adotada uma implementação manual para a serialização e desserialização das mensagens transmitidas. Sabendo do tipo dos campos e da quantidade de bytes que eles devem ocupar, foram criadas classes auxiliares que recebem estes campos e realizam as transformações necessárias. A transformação de strings para bytes foi trivial, uma vez que cada caractere da string já ocupa somente 1 byte.

Uma decisão de implementação precisou ser tomada na transformação de inteiro para bytes. A decisão foi por utilizar a ordenação de bytes big endian, e tanto o cliente quanto o servidor precisam serializar e desserializar inteiros nas mensagens respeitando esta ordem de bytes. Não foi encontrada nenhuma referência que mostrasse qual seria o "melhor" padrão, então a escolha foi arbitrária.

Uma vez definidas as representações dos campos em bytes, os bytes resultantes foram simplesmente concatenados seguindo a ordem estabelecida na especificação. Nesta fase, o cliente e o servidor trocam as mensagens básicas de controle excluindo ACK.

Canal de Dados
Uma vez que foi implementado um canal de controle estável, o desenvolvimento passou para o canal de dados. Foi acatada a sugestão dos professores de construir esta parte aos poucos.

Primeiramente, foi feita a transmissão do arquivo em uma única mensagem pelo socket UDP. O servidor receberia a mensagem FILE, extrairia os bytes relativos ao conteúdo arquivo, escreveria o arquivo no sistema de arquivos, e enviaria a mensagem de ACK. Como somente uma mensagem FILE foi transmitida, o número de sequência é 1, e o tamanho esperado do payload é o tamanho do arquivo que já era sabido previamente.

Após esta transferência básica, o próximo passo foi implementar uma estratégia simples de retransmissão: stop and wait. Nesta fase, foi implementado o requerimento de tamanho máximo do payload (1000 bytes) para mensagens FILE. A estratégia é simples: o arquivo é dividido em partes de no máximo 1000 bytes. Em sequência, uma parte é enviada e a mensagem ACK para esta é esperada imediatamente depois. Um timeout arbitrário de 10 segundos foi estabelecido, e caso ocorresse, a transmissão era feita novamente.

Neste ponto foi encontrada a dificuldade de saber o tamanho da mensagem que está no socket UDP. O tamanho do payload é de no máximo 1000 bytes, então podem ter payloads de 523 bytes por exemplo. Do lado do servidor, na leitura do buffer do socket é preciso especificar a quantidade de bytes que deve ser lida. A solução encontrada foi ler os

primeiros bytes no buffer em modo "peek", que retorna os bytes sem removê-los do buffer. São lidos todos os bytes da mensagem FILE exceto o campo de payload. Nestes bytes, pode ser encontrado o tamanho do payload da mensagem. Sabendo o tamanho do payload, agora podem ser lidos todos os bytes da mensagem juntamente com o payload, removendo do buffer.

Depois de implementar o stop-and-wait, o sistema foi modificado para utilizar a janela deslizante go-back-n. A razão para a escolha deste tipo de janela foi por aparentar ser mais simples do que a retransmissão seletiva. Neste tipo de janela deslizante, o cliente envia N pacotes para o servidor antes de parar para esperar por uma mensagem ACK. São mantidos 2 apontadores: um para o último pacote enviado, e um para o último pacote cuja confirmação foi recebida. Caso haja algum erro na transmissão de um pacote, o cliente volta a transmitir a janela começando novamente pelo último pacote enviado que não teve confirmação, e transmite novamente toda a janela.

Para implementar o go-back-n, o cliente foi alterado para criar 2 threads, sendo que uma é para o envio de mensagens FILE e a outra é para o recebimento de mensagens ACK. Um objeto que guarda os apontadores da janela deslizante é compartilhado pelas duas threads. Este objeto expõe métodos que lidam com os problemas de acesso concorrente destas duas threads, criando locks para os dois campos. Quando o cliente recebe a mensagem OK do servidor, as duas threads são criadas, e o processo principal aguarda pelo término da execução de ambas antes de realizar o envio das outras mensagens de finalização.

O principal desafio foi depurar o programa para encontrar bugs nos apontadores da janela deslizante, visto que estes são atualizados por ambas as threads. Uma vez acertada a atualização dos apontadores e os momentos de parada, a transmissão do arquivo ocorreu sem problemas.

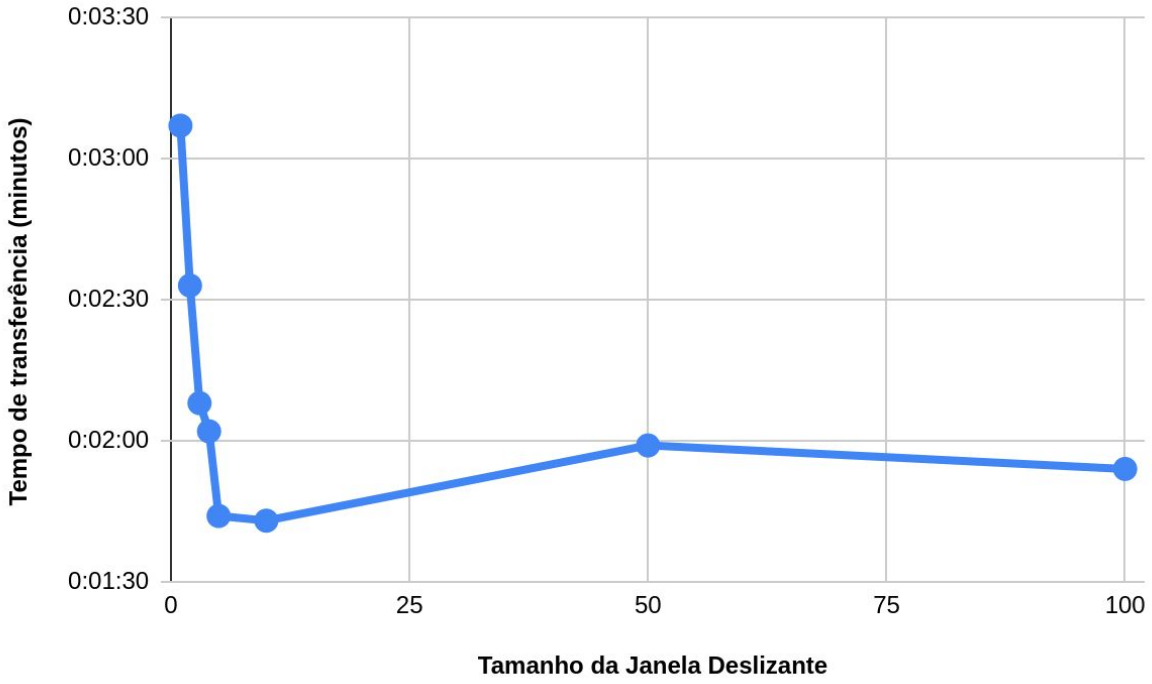
Emulador do estado da rede

Foi utilizada a ferramenta para controle de tráfego TC. Praticamente todos os comandos fornecidos na especificação funcionaram sem grandes mudanças, porém uma dificuldade encontrada foi a de como utilizar a interface dummy para a comunicação entre o cliente e o servidor. Este problema foi contornado fazendo a alteração da interface "lo" (loopback), que já era utilizada durante as execuções anteriores.

Esta ferramenta foi crucial para a descoberta de bugs na implementação da janela deslizante. Os bugs foram corrigidos, mas ainda assim o sistema estava demorando muito para fazer o envio do arquivo. O problema na implementação era o tempo de timeout arbitrário de 10 segundos, que foi colocado na implementação inicial do stop-and-wait.

Para resolver este problema, foi utilizada a técnica de retransmissão adaptável, que consiste em estimar o round trip time (RTT), e atualizar a estimativa de acordo com o tempo gasto nas transmissões seguintes. O valor inicial do RTT foi colocado arbitrariamente em 1 segundo, e foi feita uma suavização da média dos tempos, com fator alpha de 1 / 8. Se um timeout ocorrer, o valor utilizado para atualização é o próprio tempo esperado para o timeout. Assim, a média é ajustada para se aproximar do RTT do próximo pacote.

Experimento
Foram coletados os tempos de transferência de um arquivo de vídeo, de tamanho 1,6 megabytes. O parâmetro avaliado foi o tamanho da janela deslizante. Os valores utilizados para tamanho da janela foram: 1, 2, 3, 4, 5, 10, 50, 100. Foram utilizadas as configurações sugeridas na especificação para alteração do comportamento da interface de rede. No gráfico abaixo, podem ser vistos os resultados:



Pode-se perceber que o aumento do tamanho da janela deslizante diminui o tempo de transferência do arquivo. Porém, o ganho tem um limite que é rapidamente atingido com uma janela de tamanho 10. Outros parâmetros poderiam ter seus valores alterados para obter ganhos, como o fator de atualização da estimativa do RTT ou os próprios parâmetros que emulam o comportamento da interface de rede. Acredita-se que haja um balanço entre o tamanho da janela e a estimativa de RTT para cada comportamento de rede.

Pontos a melhorar
Realização de mais experimentos para validação das hipóteses elaboradas; sinalização entre as threads de envio e recebimento de mensagens no cliente; comparação dos diferentes tipos de janela deslizante; experimentos para avaliação do impacto na performance quando múltiplos clientes estão conectados; não carregar o arquivo todo em memória.

Conclusão
O trabalho permitiu o entendimento dos desafios na implementação de sistemas de transferência de arquivos, bem como outros que utilizam o protocolo UDP e implementam garantias para consistência.. Pode ser estudado o paradigma de comunicação servidor-cliente, o estabelecimento de um protocolo, programação concorrente, e compartilhamento de memória entre processos.