Trabalho prático 3

Redes de computadores

Aluno: Eduardo Ávila Vilar
Matrícula: 2017001796

26/12/2022

Universidade Federal
de Minas Gerais
– · –
Instituto de Ciências Exatas
Ciência da Computação

Relatório do trabalho prático 1

Terceiro Relatório da disciplina de Redes de Computadores do Departament ode Ciência apresentado no curso de Engenharia de Sistemas da Universidade Federal de Minas Gerais.

Aluno: Eduardo Ávila Vilar

Professor: Dorgival Olavo Guedes Neto

Dezembro
2022

Sumário

1 Introdução

O trabalho prático 3 da disciplina de Redes de Computadores consiste em implementar, em Python, um chat entre duas ou mais aplicações clientes onde todas as mensagens passam por um programa servidor antes. Todas as mensagens são confirmadas após serem enviadas. Clientes novos devem enviar uma mensagem "oi" e ao sair, uma mensagem "flw". Tanto o cliente quanto o servidor devem usar select para administrar múltiplas possibilidades que podem ocorrer concomitantemente de forma que uma não bloqueie a outra. No caso do servidor, ele deve estar preparado para receber mensagens por qualquer dos soquetes dos clientes, bem como enviar mensagens de volta para eles. Já os clientes devem ser capazes de enviar mensagens para outros clientes, receber mensagens do servidor e exibir outras mensagens na tela.

O programa deve ser construído sobre o TCP e funciona com base em 5 tipos de pacotes: tipo *ok*, que pode ser enviado tanto pelo servidor quanto pelo cliente e serve para confirmar que alguma mensagem foi recebida. Tipo *erro*, enviada somente pelo servidor para indicar que embora a mensagem tenha sido recebida, o seu objetivo não foi cumprido, por exemplo, quando um cliente não consegue se conectar ao servidor. Tipo *oi*, enviada pelos clientes para o servidor tão logo quando uma conexão é estabelecida entre esses dois. Funciona, portanto, como um 2-way handshake. Tipo *flw*, também enviada pelo cliente para o servidor, visando avisar este último sobre a saída do primeiro. E tipo *msg*, enviada pelos dois e tipo principal do programa. Carrega consigo uma mensagem para um cliente ao para todos eles, sendo central para a comunicação em geral no chat.

Todos os tipos têm quatro campos pelo menos, todos de dois bytes: tipo, sendo um dos citados acima. Transmissor, sendo um número que o cliente escolhe ao se conectar, ou 65535, no caso do servidor. Receptor, sendo o número do cliente o qual se destina a mensagem. Número de sequência, essencial para as confirmações de cada mensagem. Para mensagens do tipo "msg", dois campos a mais deve ser adicionados. O primeiro trata-se do tamanho da carga. O segundo é a carga em si, ou seja, o texto que deve chegar ao outro lado.

2 Arquitetura

O programa foi pensado para ser feito em três partes: `cliente.py`, `servidor.py` e `comum.py`.

O arquivo comum implementa objetos e funções que são comuns a ambas partes da aplicação, cliente e servidor. Por exemplo, um objeto `Protocol` responsável por manter o número de sequência das mensagens dos clientes, por impedi-los de usar funções destinadas somente ao servidor, e por ajudar ambos a criar pacotes no formato certo e já em bytes, para que possam ser enviados e recebidos pelo soquete. Neste arquivo ainda se encontra uma função responsável

por extrair as informações de pacote independente do seu tipo.

O cliente começa com as restrições básicas: seu identificador deve ser um número entre 1 e 65535 e ele deve enviar uma mensagem do tipo "oi" assim que se conectar com o servidor. Ele mantém 2 filas: uma para as mensagens que ele já escreveu e devem ser mandadas pelo seu soquete, outra para as mensagens recebidas que devem ser exibidas em sua tela. Na função select, três arquivos são mantidos sobre observação: o seu soquete, responsável pela comunicação com o servidor e, por consequência, com os outros cliente. Sua entrada *stdin* e sua saída *stdout*. O primeiro deve coletar as mensagens que o cliente escreve e colocá-las na fila para serem enviada através do soquete, enquanto o segundo coleta as mensagens na fila recebidas pelo soquete e as exibe na tela.

Já o servidor, começa criando o seu soquete sem host e com porta 5555 e o deixa no modo escuta, de forma que todos possam se conectar a ele. Um dicionário é utilizado visando ligar a cada chave numérica, um soquete de um cliente, sendo a primeira o seu identificador. Em seguida, ele cria duas listas e uma fila. Uma lista com todos os soquetes que ele deve observar esperando entradas e um com todos que ele deve vigiar aguardando para enviar algo por eles. A fila guarda as mensagens recebidas por ele, mas que precisam ser reencaminhadas. Dentro do select, o seu próprio soquete fica à espera de novos clientes e de seus respectivos ois. Quando esse procedimento inicial é cumprido, ele os coloca na lista de observação de entradas, já que o servidor nunca responde sem ser perguntado antes. Caso haja um erro na requisição do cliente, como o caso em que um cliente tenta se identificar com um número que já está sendo usado por outro, uma mensagem do tipo "erro" é enviada nessa fase também. As mensagens dos clientes que querem se comunicar com outros ou se desligar do sistema são recebidas no próximo passo, onde os soquetes que não são o soquete do próprio servidor são observados. Caso um cliente envie uma mensagem, esta é conferida de forma a saber para quem se deseja reencaminhar. Depois disto, ela é colocada na lista de envios para ser entregue ao cliente destino no próximo passo. Depois de observar todos os soquetes da lista de receptores, as lista de transmissores é observada. Nela, todos os soquetes que têm mensagens a serem enviadas são usados para acabar com essa pendência. Caso a mensagem não consiga passar pelo soquete, isso é tratado como um erro com o soquete e ele é desconectado do sistema. Seu soquete é retirado das listas de recepção e transmissão, além do dicionário de clientes.

3 Comum

O código de começa por implementar um função `extract_info(data: bytes)` que recebe como argumento uma sequência de bytes e extrai, a princípio, 4 informações dele: o tipo de mensagem, o número do remetente e do destinatário, respectivamente, e o número de sequência da mensagem. Caso a mensagem não seja do tipo "MSG", essas são as informações retornadas pela

função. Caso contrário, ela ainda extrai o tamanho da mensagem e a mensagem em si.

Neste mesmo arquivo `.py`, há uma classe implementada: `Protocol`. Essa classe contém as constantes utilizadas na comunicação em formato de bytes. Por exemplo, a constante `ok` guarda o valor `b'0001'`, pois a mensagem do tipo "ok" tem sua numeração igual a 1. Já a constante `server` vale 65535 em bytes, pois essa é a identificação do servidor.

A função inicializadora deve receber como argumento o remetente e pode receber também o destinatário. No caso deste último não ser fornecido, ela assume que o destinatário é o servidor. De qualquer forma, sua função é apenas atribuir estes argumentos a atributos internos a função que guardam estes valores.

Caso se deseje enviar uma mensagem que não seja para o servidor, existe a função `change_receiver` que deve ser chamada antes e recebe apenas o identificador do novo destinatário como argumento.

Por fim, o método mais importante da classe é o `make_frame`. Ele precisa receber o tipo de mensagem que deve ser criada, e lhe pode ser atribuído os argumentos número de sequência e mensagem. O número de sequência só pode ser utilizado pelo servidor, já que ele nunca inicia uma conversa sem ter recebido nada antes, e pelo cliente quando envia uma mensagem do tipo "ok". Essa função cria um quadro em bytes com base nos argumentos e atributos da instância.

4 Cliente

O programa cliente começa por implementar a função `msg_validation`, responsável por garantir que uma mensagem ou inicia com um "M" seguido de um número, ou inicia com "S".

Em seguida começa o programa principal. O primeiro a ser feito, é garantir que lhe está sendo dado tantos argumentos quanto necessário, ou seja, pelo menos 3: seu identificador, host e porto. Caso o identificador seja maior do que 65534 o programa pede um novo identificador ao cliente.

Logo, ele cria um socket com estas informações e envia uma mensagem do tipo "oi" através de uma instância de `Protocol`. Ao obter uma resposta positiva, ele cria duas filas, uma para os dados que vêm de `stdin` e outra para os que devem ir para `stdout`. Também cria duas listas que ficam sob a supervisão do `select`. Um com o `stdin` e outra com o `stdout`, ambas com o socket novo estabelecido com o servidor.

Dentro do `while` que envolver as listas do `select`, a primeira a ser vista

é a lista do `writable`. Nela, o programa confere se a fila de recepção a serem exibidas está vazia ou não e se o arquivo em questão é o `stdout`. Caso haja algo na fila e seja ele mesmo, ele as exibe na tela. Uma rota alternativa confere se não é o arquivo de output e se há algo na fila de envio. Em caso afirmativo, o socket em questão as envia.

Na lista `readable`, o `stdin` recebe uma mensagem do cliente e garante que ela está em um formato correto. Caso não seja o `stdin` o socket recebe a mensagem e a processa, enviando para a fila de recepção para serem exibidas na tela.

No momento em que um erro acontecer, o programa fecha o socket defeituoso e se encerra.

5 Servidor

O servidor, diferentemente do cliente, tem um dicionário para guardar a relação entre o identificador do cliente e seu socket. A princípio, o sua lista de arquivos *readable* só tem o seu próprio socket, que tem que ficar constantemente sobre observação para aceitar novas conexões.

Já sob a guarda da função `select`, o seu socket fica a espera de novas conexões. Tão logo uma nova é aceita, ele espera a mensagem de "oi" deste cliente e caso ele não use um identificador já no dicionário, ou o servidor não esteja lotado (i.e., com mais de 255 conexões), ele insere o seu identificador no dicionário e envia de volta uma mensagem de "ok". Senão, uma do tipo "erro". Uma fila para este socket também é criada.

Se o arquivo não for o próprio socket do servidor, é porque se trata de um socket de um cliente. Nesta linha, é esperado receber dados. Havendo confirmação, uma mensagem de "ok" é colocada na fila deste socket. Também, se tratando de um tipo "msg", a mensagem recebida é colocada na fila do destinatário. Caso o destinatário seja 0 (zero), ela é colocada na fila de todos do dicionário. Se for um tipo "flw", a resposta de "ok" vem instantaneamente, seguida do fechamento do socket e de sua exclusão das listas do `select` e do dicionário.

Nos sockets da lista `writable`, ele apenas tenta enviar as mensagens que estão na fila dele. Se não conseguir, o socket é desligado. Na lista de `exceptional`, todos são desligados, pois chegar neste ponto indica que o socket está defeituoso.

6 Discussão

O programa, de forma completa, não foi concluído. Principalmente, houveram dificuldades com a função `select` no cliente e sua integração com o `stdin`.

O problema maior é que no caso em que mensagens chegam ao cliente, elas só são exibidas quando este envia uma mensagem, como se o programa esperasse uma reação do `stdin` para dar prosseguimento. No mais, tudo funcionou como deveria.

De qualquer forma, foi muito valioso aprender a usar o protocolo TCP e esta última função, embora ainda reste a este aluno aprender mais sobre como ela funciona e como integrá-la com a entrada e saída padrão do sistema na linguagem `python`. Um trabalho divertido, porém não trivial.