Relatório do Trabalho Prático 03

Redes de Computadores

Thiago Martin Poppe

25/08/2021

1 Uma breve descrição

1.1 Problema proposto

Desenvolvimento de programas para compor um sistema simples de troca de textos e arquivos multimídia como áudios, vídeos e imagens. Nesse sistema teremos 3 entidades, sendo elas o emissor, que apenas envia mensagens, o exibidor, que apenas recebe e exibe as mensagens, e o servidor, que fará o processo de redirecionamento e validação das mensagens enviadas de um cliente emissor para um cliente exibidor.

Foi definido um protocolo a ser seguido, onde teremos no total 9 tipos de mensagens possíveis, onde todas terão 8 bytes de cabeçalho, possuindo a seguinte configuração:

- Os primeiros 2 bytes serão reservados para o tipo de mensagem, podendo ser do tipo:
  - i OK: mensagem de confirmação para retornarmos que uma mensagem foi recebida com sucesso. Mesmo utilizando o protocolo TCP, devemos esperar por uma confirmação (OK ou ERRO) após cada mensagem enviada.
  - ii ERRO: mensagem de confirmação para retornarmos que houve algum erro na mensagem enviada.
  - iii OI: primeira mensagem enviada para estabelecermos uma conexão entre o cliente e exibidor. Nessa mensagem podemos informar se iremos realizar uma associação entre um emissor e um exibidor.
  - iv FLW: mensagem enviada para o servidor para informar a desconexão de algum cliente. No caso do sistema implementado, os clientes podem encerrar sua comunicação apertar `Ctrl + C` ou digitando FLW (no caso do emissor).
  - v MSG: mensagem para enviarmos uma informação textual em ASCII. Essa mensagem pode ser enviada para todos os exibidores do sistema (`broadcast`).
  - vi CREQ: mensagem enviada para solicitarmos uma lista de clientes conectados no sistema. Essa mensagem pode ser enviada para todos os exibidores do sistema (`broadcast`).
  - vii CLIST: mensagem contendo uma lista dos clientes conectados no sistema. Essa mensagem será enviada somente pelo servidor para um ou mais exibidores, como informado pela mensagem CREQ.
  - viii FILE: mensagem para iniciarmos uma comunicação de arquivo multimídia. Essa mensagem pode ser enviada para todos os exibidores do sistema (`broadcast`).
  - ix FILE_CHUNK: mensagem contendo um "pedaço" (*chunk*) do arquivo propriamente dito.
- Os próximos 2 bytes serão o identificador de origem da mensagem. Usaremos esse campo para validarmos se algum cliente não está se passando por outro cliente.
- Os próximos 2 bytes serão o identificador de destino da mensagem. Os emissores terão identificadores entre 1 e $2^{12} - 1$; os exibidores entre 2^{12} e $2^{13} - 1$; e o servidor terá identificador fixo igual à $2^{16} - 1$. Para enviar uma mensagem para todos os exibidores (`broadcast`), esse campo deve ser igual à 0.
- Os últimos 2 bytes serão reservados para o número de sequência da mensagem. Onde cada mensagem enviada (exceto mensagens de confirmação) terão o seu próprio número de sequência.

1.2 Observações

- Todo o sistema foi desenvolvido utilizando a linguagem de programação Python, especificamente a versão 3.8.5. Além disso, foi utilizado a biblioteca `socket`, nativa da linguagem, para a programação com sockets e toda a comunicação das pontas, que foi feita utilizando TCP com protocolo IPv4.

- O sistema desenvolvido assume que há somente uma associação 1:1 entre emissores e exibidores, isto é, um emissor só pode se associar a um único exibidor e vice-versa. Após encerrada a comunicação de um emissor, o seu exibidor associado será encerrado; se um exibidor for encerrado, o emissor continuará funcionando normalmente. Infelizmente não há uma forma de notificar o emissor de que o seu exibidor foi desconectado, portanto, apenas printei no terminal do servidor um "log" informando o processamento dessa mensagem.

2 Visão geral da implementação

Para o estabelecimento da comunicação com múltiplos clientes foi utilizada a função `select` como sugerido pelo enunciado do trabalho prático. Para sabermos de qual cliente é um `socket`, optei por manter um dicionário que mapeia um `socket` para um identificador, inclusive mapeando o `socket` do próprio servidor com o seu identificador.

Para manter o código mais limpo, foi criado 3 funções para processar as informações referentes à cada `socket` do sistema. No caso, quando estivermos trabalhando com o `socket` do servidor, iremos chamar a função `create_client`, inicializando assim um novo cliente no sistema. O mesmo é feito para os clientes, chamando a função `process_sender` e `process_displayer` contendo a lógica para processarmos um emissor e exibidor, respectivamente.

O armazenamneto dos clientes foi feito através de um dicionário para cada tipo de cliente, guardando informações como o seu `socket` e se temos alguma associação para aquele cliente, por exemplo quando temos um emissor associado com um exibidor (e vice-versa).

É importante notar que foi implementado um sistema dinâmico para a atribuição do identificador para um novo cliente do sistema, mantendo dois conjuntos (`set`) de identificadores possíveis, um para os emissores e outro para os exibidores. Quando um novo cliente é inicializado, realizamos um `pop` no conjunto referente àquele cliente e quando ocorre uma desconexão nós realizamos um `add` no conjunto, permitindo assim reutilizar o ID para outro cliente. O servidor não irá aceitar mais clientes uma vez que esses conjuntos estiverem vazios, isto é sem identificadores disponíveis, retornando assim uma mensagem de erro.

3 Implementação do emissor

A interface do emissor foi definida de forma que uma mensagem `OI` fosse mandada de forma automática para o servidor. Ao recebermos um OK como resposta, iremos desejar as boas-vindas ao cliente e exibir o seu ID juntamente com as instruções para mandar mensagens e se comunicar no sistema.

As mensagens permitidas pela interface serão `FLW`, não assumindo parâmetros e sempre sendo enviada para o servidor; `MSG` que recebe como parâmetro o ID do destinatário e a mensagem propriamente dita em ASCII; `CREQ` que recebe como parâmetro apenas o ID do destinatário; e por fim `FILE` que recebe como parâmetro o ID do destinatário e o `nome.extensão` do arquivo multimídia em ASCII. Note que no caso de uma mensagem `FILE`, todos os arquivos enviados devem estar na pasta `enviar`, apenas para deixar os arquivos mais organizados. De forma similar ao trabalho prático anterior, iremos maximizar o tamanho de um `chunk`, ou seja, sempre iremos mandar blocos de tamanho igual à $2^{16} - 1$ bytes quando possível, evitando assim fragmentações excessivas e desnecessárias.

A interface fornece um sistema simples, porém robusto, de tratamento de exceções, informando ao usuário possíveis erros de formatação da mensagem. Caso alguma exceção inesperada ocorra, a interface irá automaticamente desconectar o cliente enviando uma mensagem de `FLW` para o servidor.

4 Implementação do exibidor

A interface do exibidor será relativamente simples, tendo a mesma forma de inicialização que o emissor. Note que diferentemente do emissor, o exibidor não aceita nenhuma entrada do teclado (exceto o `Ctrl + C` solicitando a sua desconexão). Sendo assim, sua implementação foi bem mais simples, não possuíndo por exemplo um tratamento de exceções como foi feito no emissor.

A interface irá sempre exibir as mensagens enviadas no formato `<type> from <id>: <message>`, ou seja, o tipo da mensagem, o id de origem (emissor) e a mensagem propriamente dita, podendo ser o nome de um arquivo, uma lista de clientes ou uma mensagem textual.

Todos os arquivos multimídia recebidos por um exibidor serão salvos na pasta `recebidos`, apenas para deixar os arquivos mais organizados. Além disso, os arquivos serão salvos seguindo o formato: IdExibidor_IdEmissor_IdArquivo.EXT.

5 Implementação do servidor

Como mencionado anteriormente, o servidor irá manter algumas estruturas de dados para manipularmos os `sockets` conectados no sistema bem como a atribuição dos identificadores para os clientes seus clientes.

De forma geral, o servidor construído será robusto à erros, verificando sempre as mensagens recebidas e "logando" cada processamento feito. Iremos encaminhar as mensagens para os respectivos exibidores caso a mensagem esteja correta ou retornaremos um erro para o cliente de origem. É importante notar que caso algum comportamento inesperado aconteça, por exemplo algo que não deveria acontecer dado o protocolo, o servidor pode ter um comportamento inesperado, possivelmente não aceitando mais mensagens. Optei por não tratar esses casos uma vez que o código iria ficar muito mais complexo e extenso do que já é. Sendo assim, apenas as funcionalidades solicitadas, bem como funcionalidades extras (como o `broadcast` de mensagens `CREQ` e `FILE`) foram implementadas.

5.1 Criação de um novo cliente

Como mencionado anteriormente, foi criado uma função chamada `create_client` para lidarmos com a criação de um novo cliente no sistema. Além de verificarmos se possuímos algum identificador disponível, apenas iremos esperar receber mensagens do tipo `OI`. Caso algum outro tipo de mensagem seja enviado, iremos retornar um erro para o cliente de origem.

A mensagem do tipo `OI` deve ser a primeira mensagem enviada para estabelecermos uma comunicação com o cliente. Sendo assim, verificaremos se o cabeçalho dessa mensagem possui o tipo `OI`, se o destinatário da mensagem realmente é o servidor e se o número de sequência dessa mensagem é 0 (número de sequência inicial). Caso essa validação seja correta, iremos retornar uma mensagem de `OK` contendo o identificador do cliente no campo de destino da mensagem.

5.2 Processamento do socket de um emissor

Foi criado a função `process_sender` para processarmos as mensagens enviadas por um emissor. Essa função é a mais complexa de todo o servidor, ocupando boa parte do código fonte desenvolvido. Para cada tipo de mensagem foi criado uma função para processar essa mensagem a fim de mantermos o código o mais sucinto e limpo possível.

5.2.1 Mensagens do tipo FLW

As mensagens do tipo `FLW` irão modificar as estruturas de dados utilizadas para armazenar os clientes e identificadores disponíveis. Caso um emissor envie uma mensagem de `FLW`, iremos primeiro desconectar o seu exibidor associado (caso exista) enviando também uma mensagem de `FLW` para ele. Optei por desconectar o emissor somente se conseguirmos desconectar primeiro o seu exibidor já que isso representa uma lógica mais plausível do que seria feito em um sistema real, mesmo que um erro não seja possível no protocolo definido para esse trabalho.

5.2.2 Mensagens do tipo MSG e CREQ/CLIST

As mensagens do tipo `MSG` e `CREQ` possuem o mesmo formato de processamento, onde iremos recuperar a mensagem textual (no caso do `MSG`) ou uma lista de clientes conectados ao sistema (no caso do `CREQ`) bem como uma lista de exibidores aos quais devemos enviar essa mensagem. A lista de exibidores será recuperada através da função `get_send_to_list`, que irá através do campo de ID do destino, retornar se devemos encaminhar a mensagem para todos os exibidores, um exibidor específico ou um exibidor associado a um emissor. Por simplicidade optei por sempre retornar uma lista a fim de manter o código único para cada um desses casos. Caso haja um erro no ID de destino, ou seja um ID inválido, essa função irá retornar `None`, indicando que não conseguimos mandar a informação para nenhum cliente exibidor, o que faz retornar um erro para o cliente de origem.

5.2.3 Mensagens do tipo FILE e FILE_CHUNK

Essas mensagens foram processadas de forma similar às mensagens anteriores, onde também iremos recuperar uma lista de exibidores aos quais devemos encaminhar as informações passadas para o servidor. Usaremos o cabeçalho "estendido" da mensagem `FILE` para recuperar metadados do arquivo, como a sua extensão, número de `chunks` esperados e ID do arquivo. Para cada exibidor na lista de exibidores "`send_to`", iremos encaminhar a mesma mensagem `FILE` passada para o sevidor, informando assim que iremos iniciar o processo de transmissão de `FILE_CHUNKS`. Para cada `chunk` enviado, nós devemos esperar uma confirmação da outra ponta para que possamos enviar o próximo `chunk`.

5.3 Processamento do socket de um exibidor

O processamento do socket de um exibidor será feito através da função `process_displayer`, e ela terá a única função de desconectar o exibidor do sistema uma vez que receber a mensagem `FLW`.

6 Observações gerais

6.1 Organização do código

Para o desenvolvimento desse trabalho prático, criei 3 arquivos fonte, sendo eles `server.py`, `sender.py` e `displayer.py` contendo a lógica do servidor, interface do emissor e interface do exibidor, respectivamente.

Além disso, foi desenvolvido um módulo chamado `utils` que contém alguns pacotes com algumas funções e constantes utilizadas durante o desenvolvimento do trabalho, como por exemplo o tamanho do buffer a ser lido por cada socket, funções de criação de mensagens e algumas funções em comum como é o caso da função que inicializa a interface do cliente (tanto emissor como exibidor).

Cada programa irá receber alguns argumentos na linha de comando. No caso do servidor, será passado o número da porta que ele deve "ouvir"; no caso do emissor, devemos informar o IP e porta no formato `IP:porta` e opcionalmente um exibidor associado; e por fim para o exibidor nós devemos informar o IP e porta no mesmo formato apresentado para o emissor.

6.2 Testes e verificações

Para esse trabalho prático foi desenvolvido alguns testes manuais, enviando diversos tipos de arquivos como vídeos, fotos e áudios em diferentes formatos e tamanhos.

7 Conclusão

De forma geral, esse trabalho prático permitiu de forma bastante interessante a aplicação de toda a teoria vista em sala de aula através de um exemplo prático. Diferentemente do trabalho prático anterior, este possuiu um protocolo mais simples, permitindo com que o aluno fosse capaz de implementar um sistema em mais alto nível, sem se preocupar muito com aspectos relacionados à correção e verificação da integridade das mensagens, como foi o foco do trabalho prático anterior.