MC833 - Projeto 1

Carlos Avelar (168605) e Tiago Loureiro Chaves (187690)

04 de Abril, 2019

1 Introdução

Este projeto teve como objetivo a implementação de um sistema cliente-servidor com comunicação baseada no protocolo TCP utilizando sockets, na linguagem de programação C.

O sistema baseia-se em uma rede de perfis profissionais, tendo opções de busca e alteração de dados oferecidas pelo servidor (p.e. inserção de experiências profissionais, listagem de perfis de determinado curso, etc.), as quais são mostradas ao cliente, que as pode requisitar indefinidamente (em um laço), até que decida fechar a comunicação.

Assim, os dados enviados pelo cliente são exclusivamente textuais, enquanto que o servidor também pode enviar arquivos de imagem `.png`, com as fotos dos perfis, além de texto.

Fez-se também uma análise de tempo médio das opções, considerando tanto seu tempo de execução, no servidor, como o tempo total de comunicação observado no cliente.

2 Sistema

Uma visão geral do sistema cliente-servidor pode ser facilmente obtida a partir do diagrama abaixo, que traz uma representação em alto nível do seu fluxo de execução:

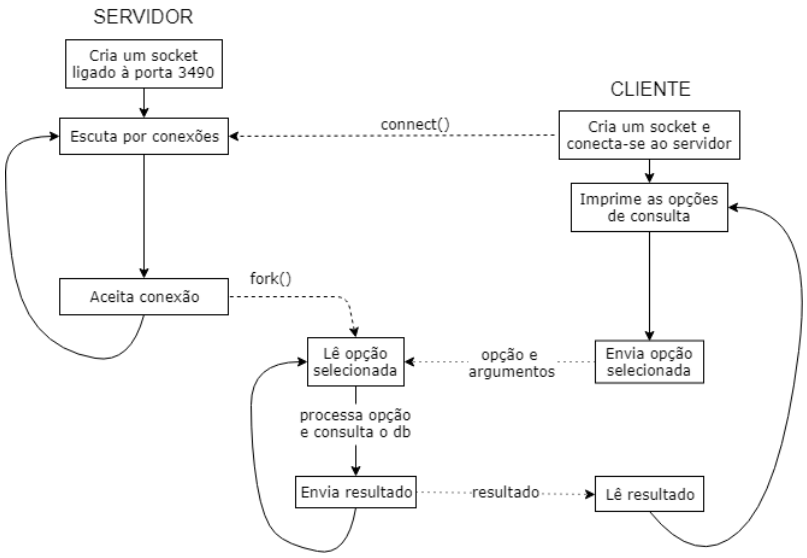

Figura 1: Diagrama de funcionamento do sistema

Primeiramente, deve-se iniciar o servidor, executando-se `./server` (após gerados os arquivos executáveis, por exemplo com `make all`, estando no diretório do `Makefile`). Uma vez estabelecida a ligação (*bind*) com a porta 3490, o servidor escuta por conexões de clientes.

Analogamente, iniciamos o cliente com `./cliente <hostname>`, onde `hostname` é o nome ou endereço da máquina onde foi executado o servidor (p.e. `localhost` se ambos estiverem rodando na mesma).

Em seguida, o cliente conecta-se ao servidor e imprime para o usuário a seguinte lista de opções:

- (1) listar todas as pessoas formadas em um determinado curso;
- (2) listar as habilidades dos perfis que moram em uma determinada cidade;
- (3) acrescentar uma nova experiência em um perfil;
- (4) dado o email do perfil, retornar sua experiência;
- (5) listar todas as informações de todos os perfis;
- (6) dado o email de um perfil, retornar suas informações;
- (7) sair.

Assume-se então que o usuário digita corretamente o número da opção que deseja e possíveis argumentos que elas necessitam (como o curso desejado, na opção 1).

O cliente envia portanto as informações necessárias ao servidor, o qual consulta o banco de dados (`db`) e envia o resultado de volta ao cliente.

Os resultados são *printados* no terminal do cliente, e salvos no arquivo `option_results.txt`, que mantém um registro de todas as opções selecionadas (imagens são salvas como `<email>.png`, onde `email` refere-se ao perfil dono da foto, e é a chave da tabela no `db`).

As opções disponíveis são novamente mostradas ao cliente, e repete-se o fluxo descrito acima, até que o usuário escolha sair (opção 7).

3 Armazenamento e estrutura de dados do servidor

Optou-se pelo uso de um banco de dados `SQLite` para o armazenamento dos dados, pois assim a obtenção e alteração de informações dos perfis seria natural, com *queries* SQL.

Assim, ao executar-se o servidor é chamada a função `init_db()` que cria as tabelas necessárias, mostradas abaixo, e as popula com informações.

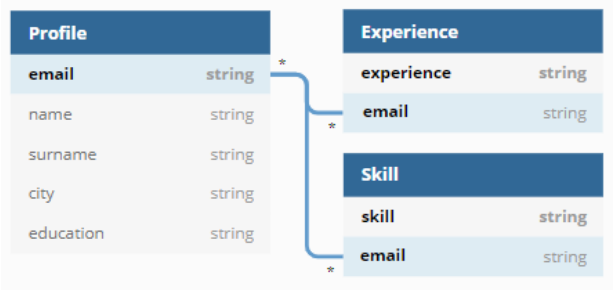

Figura 2: Tabelas do bando de dados

Os dados inseridos são os seguintes (email, nome, sobrenome, cidade, formação, habilidades e experiências):

cinco@mail.com	tres@mail.com	uno@mail.com
Cinco	Tres	Uno
Seis	Cuatro	Dos
Seattle	Campinas	Campinas
CS	CS	Linguistics
English, Reap, Sow, Spanish	Code, Read, Write	Acoustic Engineering, English
Study, Study more, Work	Study, Work	Research, Work

Nota-se que as fotos dos perfis não são guardadas no banco de dados, mas sim como imagens em uma pasta `imgs/` no mesmo diretório do arquivo executável do servidor, nomeadas `<email>.com`, sendo `email` a chave do perfil no `db` (evitando conflitos de nomenclatura).

Figura 3: cinco@mail.com.png

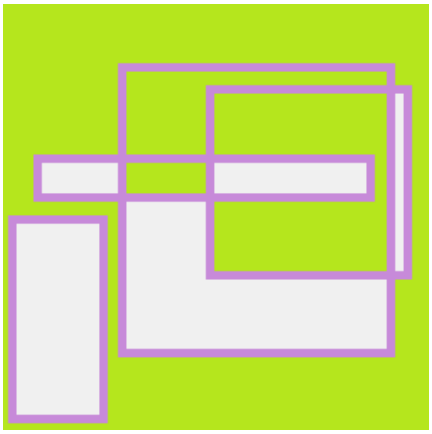

Figura 4: tres@mail.com.png

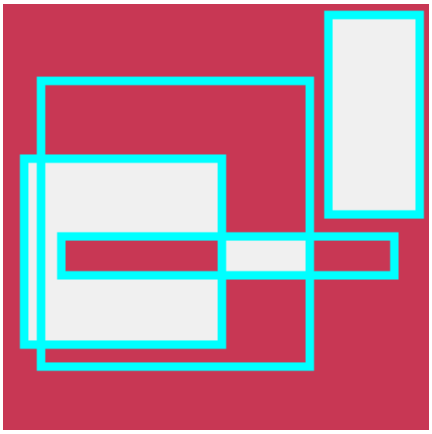

Figura 5: uno@mail.com.png

4 Detalhes da implementação do servidor TCP

Ao iniciar, o servidor cria as tabelas da Figura 2 no `db` (adicionando a elas as informações mencionadas acima). Para isso, abre-se o banco de dados com a função `sqlite3_open()` [1] e executam-se *queries* com `sqlite3_exec()` [2] com chamadas à função `execute_sql()` do servidor.

Observa-se que o resultado da última *query* realizada é sempre salvo para referência no arquivo `last_query_result.txt` (definido pela constante `FILE_SERVER` em `libs.h`).

Em seguida, o servidor cria um socket ligado à porta 3490 (definida também em `libs.h`, como `PORT`), deixa que o socket escute por conexões de clientes com a função `listen()`, e então chama `accept()` em um *loop* para aceitá-las [3].

A cada conexão aceita o processo servidor executa um `fork()`, criando uma cópia de si que irá tratar das requisições do cliente, enquanto que o processo original continua escutando por conexões, como mostrado na Figura 1, assim múltiplas conexões podem ser tratadas concorrentemente.

O processo filho segue recebendo as mensagens do cliente em um laço (veja `receive_messages()`), e invoca a função relativa à opção escolhida, dentre as listadas na Seção 2, a qual lê possíveis argumentos passados junto com a opção. São essas:

- (1) `opt_get_profiles_filtering_education()`

- (2) `opt_get_skills_filtering_city()`
- (3) `opt_add_skill_to_profile()`
- (4) `opt_get_experience_from_profile()`
- (5) `opt_get_profiles()`
- (6) `opt_get_profile()`

Se a opção escolhida é a 7 (sair), o processo filho sai do laço e fecha o socket filho.

Após lidas as mensagens do cliente, as funções acima executam as consultas/atualizações necessárias no `db` (abstraídas por funções de mesmo nome, porém com o sufixo `_sql`, que chamam `execute_sql()`). Por fim, com o resultado da *query* salvo em `last_query_result.txt`, executa-se `send_file_to_client()` para enviar seu conteúdo ao cliente.

No caso das opções 5 e 6 também é feito o envio de fotos de perfil (veja `send_picture_to_client()`), as quais estão salvas na pasta `/imgs` (criada no mesmo diretório do arquivo do servidor).

Já no cliente, após enviada a opção e lida a resposta do servidor, ela é impressa na tela (exceto pelas fotos, as quais são salvas com o mesmo nome que no servidor, i.e. `<email>.png`, vide `save_img()`). Para referência, todas as respostas de consultas são anexadas no arquivo `option_results.txt` após *printadas*, chamando-se `save_result_to_file()`.

4.1 Formato das mensagens

Uma vez que o protocolo TCP é *stream oriented* e não *message oriented*, ou seja, não há uma distinção entre grupos de mensagens, considera-se apenas um fluxo contínuo de dados/*bytes*, para que haja uma delimitação no envio e leitura de mensagens diferentes deve-se implementar um "protocolo"em cima das chamadas `send()` e `recv()` padrão TCP que faça tal divisão.

Portanto, como tanto o servidor quanto o cliente deveriam utilizar a mesma *pattern* de comunicação, criaram-se os seguintes *wrappers* no arquivo `common.c`, incluindo seu *header* em `libs.h` (que é importado por ambos):

```
int send_wrapper(int file_descriptor, char *message, int verbose);
int recv_wrapper(int file_descriptor, char **buffer, int verbose);
int send_img_wrapper(int file_descriptor, char * message, int msg_size, int verbose);
int recv_img_wrapper(int file_descriptor, char **buffer, int *size, int verbose);
```

Figura 6: *Wrappers*

Para diferenciar as mensagens enviadas, três abordagens foram consideradas [4]:

- Sempre enviar mensagens de tamanho fixo
- Enviar o tamanho da mensagem junto a ela
- Estabelecer um "marcador"para separar as mensagens

Tendo em vista que as mensagens variam desde um simples número enviado (indicando a opção escolhida pelo cliente) até arquivos de imagem, fixar o tamanho da mensagem não faria sentido, escolheu-se então adicionar ao início de cada mensagem seu tamanho (com 10 algarismos e ';' ao final), p.e.:

`Hello, World!` ⟵ *wrapper* ⟶ `0000000013;Hello, World!`

Figura 7: Conversão da mensagem "Hello, World!"de 13 *bytes*

Os *send wrappers* então prefixam o tamanho da mensagem e chamam o `send()` padrão em um *loop* até que todos os *bytes* sejam enviados.

Analogamente, os *recv wrappers*, esperando o prefixo, obtêm o tamanho da mensagem e chamam `recv()` até que o tamanho em *bytes* da mensagem seja lido, emulando desta forma a delimitação das mensagens.

5 Análise dos resultados

Executando o cliente e servidor em máquinas distintas, porém na mesma rede (no laboratório CC00 do IC3), marcou-se 40 vezes, para cada opção, os tempos total (visto pelo cliente) e de processamento no servidor, a partir dos quais podemos estimar o tempo de comunicação (vide Figura 5).

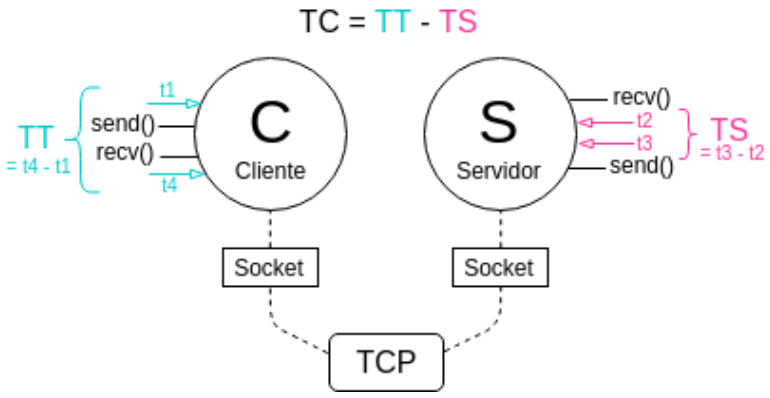

Figura 8: Tempos de comunicação (TC), total (TT) e de processamento no servidor (TS)

A tabela abaixo mostra os resultados do experimento, sendo *Avg Com. Time* o tempo de comunicação (TC) e *Avg Op. Time* o tempo de processamento por opção no servidor (TS), em micro segundos (μs). O desvio padrão e intervalo de 95% de confiança são referentes ao TC:

Op.	Avg Com. Time (us)	Std Deviation (us)	95% Conf Interval (us)	Avg Op. Time (us)
1	3874.7	405.74	3623.21 to 4126.18	6764.55
2	3884.00	303.90	3695.63 to 4072.36	10922.70
3	2832.58	280.20	2658.90 to 3006.24	2715.73
4	3903.13	408.99	3649.62 to 4156.62	6809.0
5	15494.30	696.32	15062.71 to 15925.88	8635.725
6	4584.18	400.36	4336.02 to 4832.32	5822.40

Apresenta-se nos gráficos seguintes os tempos de comunicação calculados, a média das execuções (em vermelho), e barras delimitando o intervalo com nível de 95% de confiança.

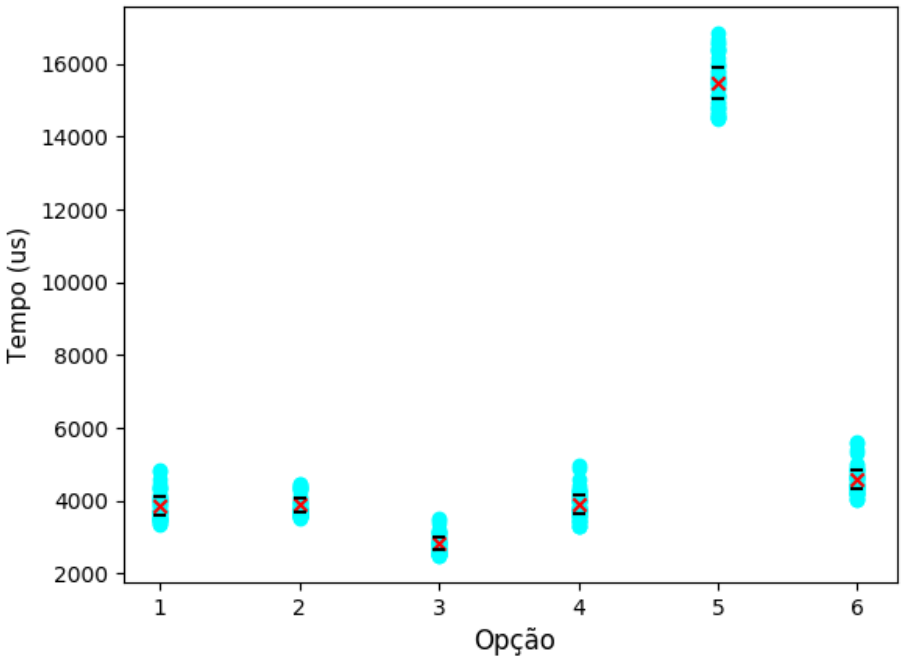

Figura 9: Gráfico dos tempos de comunicação por opção

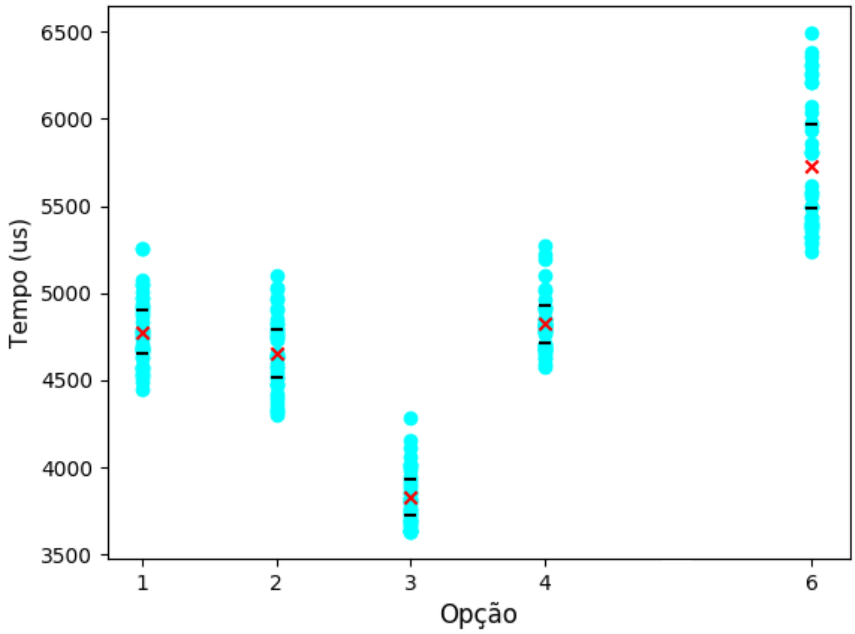

Figura 10: Gráfico ampliado para as opções 6 e de 1 a 4

Os tempos obtidos na opção 5 são os maiores pois esta é a opção que envia praticamente todos os dados do banco (lista todos os perfis).

Observamos que o tempo de comunicação da opção 3 é o mais baixo, como era esperado, pois não é feito uma consulta ao banco e sim uma inserção e, assim, a resposta ao cliente é muito curta e rápida (apenas sinaliza a inserção bem sucedida).

O código tem cerca de 1300 linhas, divididas entre 9 arquivos (além do `Makefile`):

- `libs.h`
- `common.h` e `common.c`
- `server.h` e `server.c`
- `client.h` e `client.c`
- `input_parser.h` e `input_parser.c`

6 Conclusão

Percebe-se que a decisão de projeto de fazer o armazenamento com um banco de dados `SQLite`, ao invés de salvar os dados em arquivos, simplifica a adição de novas opções (e até de informações guardadas), pois basta escrever novas *queries*, facilitando a escalabilidade.

Podemos ver que para o tipo de sistema implementado usar o TCP era sem dúvidas a opção mais recomendada, pois não foi necessário em nenhum momento se preocupar com reenvio de dados perdidos (como poderia acontecer se usássemos o protocolo UDP).

Observamos, entretanto, que criar um protocolo para comunicação em cima do TCP (que é *stream oriented*) é essencial para organizar o tratamento e diferenciação de mensagens (como descrito na seção 4.1), assim tornando-a mais simples e fácil de implementar, uma vez que a limitação entre mensagens seja abstraída, simulando um protocolo *message oriented*.

Referências

[1] "Opening a new database connection." [Online]. Available: https://www.sqlite.org/c3ref/open.html

[2] "One-step query execution interface." [Online]. Available: https://www.sqlite.org/c3ref/exec.html

[3] "A simple stream server." [Online]. Available: http://beej.us/guide/bgnet/html/multi/clientserver.html#simpleserver

[4] "Solution for tcp/ip client socket message boundary problem." [Online]. Available: https://www.codeproject.com/Articles/11922/Solution-for-TCP-IP-client-socket-message-boundary