Relatório do Trabalho Prático 01

Redes de Computadores

Thiago Martin Poppe

22/06/2021

1 Uma breve descrição

1.1 Problema proposto

Criação de um sistema simples seguindo o paradigma cliente-servidor para praticar os conceitos estudados durante as aulas teóricas. O sistema a ser desenvolvido terá 4 funções relacionadas ao processo de vacinação contra a COVID-19 e serão divididas entre os grupos **saúde** e **cidadão**:

1. O grupo saúde deve ser capaz de adicionar um novo local de vacinação.
2. O grupo saúde deve ser capaz de remover um local de vacinação.
3. O grupo saúde deve ser capaz de consultar uma lista de locais de vacinação disponíveis.
4. O grupo cidadão deve ser capaz de consultar o local de vacinação mais próximo da sua localização.

Além disso, foi definido um protocolo a ser seguido tanto pelo servidor quanto pelo cliente. O término das mensagens são dadas pelo caractere **\n** (caractere especial de quebra de linha) e as mensagens não devem ultrapassar um tamanho de 500 bytes. Caso esse protoloco seja violado por parte do cliente, o servidor é capaz de desconectar o mesmo sem interromper a sua execução.

1.2 Observações

- Todo o sistema foi desenvolvido utilizando a linguagem de programação C. Além disso, foi utilizado a biblioteca POSIX para a programação com sockets e toda a comunicação do cliente-servidor, que foi feita utilizando o protocolo TCP.
- Foi utilizado o código fonte disponibilizado pelo professor Ítalo para criar o "esqueleto" do sistema a ser desenvolvido.
- O sistema desenvolvido é 1:1, isto é, o servidor apenas consegue tratar um único cliente por vez. Quando esse cliente é desconectado, o servidor consegue aceitar novas conexões e retomar a comunicação, encerrando a sua execução apenas quando é enviando um comando `kill` por parte do cliente.

2 Estruturas de dados utilizadas

Para manipular a lista de postos de vacinação no servidor, foi utilizado um vetor simples de 50 posições (número máximo de postos), onde cada posição era representado por uma `struct Point2D_t`, que contém dois campos inteiros x e y para salvar a coordenada cartesiana daquele posto de vacinação. Toda a implementação dessa estrutura de dados, bem como suas funções, estão definidas e implementadas nos arquivos `location_array.h` e `location_array.c`, respectivamente, fornecidos juntamente com os demais arquivos.

Todas as funções implementadas por essa estrutura possuem complexidade temporal da ordem de $O(n)$, onde o "feedback" do comando executado, que será enviado como resposta para o cliente, ficará salvo em uma string que sempre será o último parâmetro das funções. Por exemplo, para adicionarmos um novo

local de vacinação, nós fornecemos (i) o nosso vetor de locais, (ii) a coordenada cartesiana do novo local de vacinação e por fim (iii) uma string que irá retornar se a criação foi feita com sucesso, se já existe esse posto de vacinação ou se atingimos o tamanho máximo de criação.

Algo interessante de se mencionar é que foi utilizado a distância euclidiana como métrica para computar o posto de vacinação mais próximo do cidadão. A distância eucliana de dois pontos é originalmente calculada através da fórmula $dist(v, w) = \sqrt{(v_x - w_x)^2 + (v_y - w_y)^2}$, onde v e w são pontos cartesianos representados pelo par ordenado (v_x, v_y) e (w_x, w_y), respectivamente. Porém, foi notado que não é solicitado a distância real entre o cidadão e a localização do posto de vacinação mais próximo, mas sim **qual** é o posto mais próximo. Com isso, para evitarmos um processamento desnecessário, podemos encontrar o posto mais próximo desconsiderando a raiz presente na fórmula da distância eucliana, evitando assim também possíveis erros relacionados à valores de ponto flutuante e aproximações truncadas. De forma geral, o problema passa a se tornar um problema de minimização, onde queremos encontrar valores de x_{min} e y_{min} tal que esses valores minimizem a função $(c_x - x_{min})^2 + (c_y - y_{min})^2$, onde o par ordenado (c_x, c_y) representa a posição do cidadão.

2.1 Funcionalidades implementadas

Como mencionado em seções passadas, o nosso servidor possui algumas funções relacionadas com a manipulação de locais de vacinação contra a COVID-19. No total, foram implementadas 4 funcionalidades que terão as suas implementações explicadas a seguir.

1. **Adição de novos locais:** para a implementação dessa funcionalidade, foi criada uma função chamada `add_location`, na qual iremos inserir um novo local de vacinação se e somente se esse local não tiver sido inserido anteriormente. A inserção propriamente dita é feita em $O(1)$ já que estamos trabalhando com um vetor e mantendo o seu tamanho. Porém, a complexidade da inserção passa a ser $O(n)$ uma vez que devemos verificar se já inserimos esse local de vacinação ou não.

2. **Remoção de um local:** para a implementação dessa funcionaldiade, foi criada uma função chamada `remove_location`, na qual iremos remover um local de vacinação se ele existir no nosso vetor de localizações. Por estarmos trabalhando com um vetor, a remoção se dá simplesmente pelo **shift** à esquerda de todos os valores à direita do valor removido. Por exemplo, para remover o elemento 4 do vetor `[1,4,3,2,5]` basta deslocarmos todos os elementos à direita do 4 uma posição para trás, resultando no vetor `[1,3,2,5]`.

3. **Consulta dos locais:** para a implementação dessa funcionalidade, foi criada uma função chamada `location_array_to_string`, na qual iremos apenas representar o nosso vetor de localizações através de uma string no formato $X_1\ Y_1\ X_2\ Y_2\ \ldots\ X_n\ Y_n$ ou como `none` caso nenhum local de vacinação tenha sido inserido ainda. Essa representação irá seguir a mesma ordem de inserção dos locais de vacinação.

4. **Consulta do local mais próximo:** para a implementação dessa funcionalidade, foi criada uma função chamada `get_closest_location` que retorna o par ordenado (x_{min}, y_{min}) mais próximo com relação à localização do cidadão ou `none` caso nenhum local de vacinação tenha sido inserido ainda. Em caso de empate, a função irá retornar o primeiro ponto encontrado dentre os mais próximos.

3 Recebimento de múltiplos pacotes e mensagens

Foi descrito no problema que as mensagens poderiam ser enviadas através de múltiplos pacotes e uma única sequência de bytes poderia ter uma ou mais mensagens. Por exemplo, a sequência de bytes "add 111 111**n**add 222 222**n**" possui duas mensagens de adição de um novo posto de vacinação.

Tanto para a leitura das mensagens no formato `cmd X Y` ou somente `cmd`, quanto para a separação das sequências de bytes em múltiplas mensagens, foi utilizado a função `strtok_r` da biblioteca padrão de C. Essa função possui a mesma lógica da função `strtok`, porém ela permite a tokenização de múltiplas strings através de um ponteiro para a string original, o que facilita a sua utilização em múltiplas seções do código com diferentes strings.

Essa foi uma das tarefas mais complicadas e um tanto quanto não trivial a princípio, uma vez que tanto o cliente quanto o servidor podem ser sujeitos à essa peculiaridade de comunicação. Foi desenvolvido soluções similares em ambas partes e elas serão discutidas com mais detalhes nas próximas seções.

3.1 Tratamento no servidor

A lógica desenvolvida por parte do servidor é de que nós sabemos que todas as mensagens devem terminar com o caractere **\n**, como foi definido no protocolo do sistema. Sendo assim, foi criado um loop para receber as partes (os pacotes) das mensagens até que a mensagem lida até então termine com o marcador de fim de mensagem.

Através da leitura da documentação da função `recv` (manual do Linux), foi possível tratar também o caso onde o cliente acaba sendo desconectado ou quando temos algum erro relacionado à comunicação. Em caso de algum erro durante a comunicação, o servidor irá parar esse loop de recepção e irá prosseguir seu fluxo normalmente, desconectando o cliente que gerou tal erro e irá esperar uma nova conexão para retomar seu fluxo de comunicação e troca de mensagens.

Uma vez que lemos uma sequência de bytes válida, isto é terminada por **\n**, o servidor faz uma repartição dessa sequência através dos caracteres de fim de mensagem presentes na sequência lida. Com isso, ele é capaz de processar tanto uma mensagem com múltiplos pacotes quanto uma sequência com múltiplas mensagens (ou até mesmo uma combinação dos dois casos).

O processamento e tratamento da entrada é feito através da função `parse_message`, onde é realizado alguns processos de verificação e validação do formato da mensagem, retornando um valor de sucesso (1) ou fracasso (0). Os valores de comando e seus parâmetros são recuperados através de ponteiros passados como parâmetros da função.

3.2 Tratamento no cliente

A lógica desenvolvida por parte do cliente é similar a do servidor no sentido de conseguir tratar a leitura de mensagens com múltiplos pacotes, uma vez que ele implementa a mesma ideia de ler a mensagem até que ela termine com um caractere de fim de mensagem.

Como o cliente é capaz de mandar múltiplas mensagens em uma única sequência de bytes, foi criado uma função chamada `count_messages` que retorna quantas mensagens existem em uma certa sequência de bytes, ou seja, essa função basicamente retorna o número de caracteres **\n** da sequência. Com isso, o cliente irá entrar em um loop até que o mesmo número de mensagens enviadas seja igual ao número de mensagens recebidas como resposta pelo servidor. Note que o servidor também pode encaminhar múltiplas mensagens em uma mesma sequência de bytes e, para resolver esse problema, o cliente irá utilizar novamente a função `count_messages` para conseguir contar o número exato de mensagens recebidas pelo servidor, atualizando o número de mensagens esperadas.

4 Observações gerais

4.1 Organização do código

O código fonte consiste de um arquivo `Makefile` que deve ser executado através do comando `make` via linha de comando, permitindo assim a compilação do código e geração dos executáveis `servidor` e `cliente`. Além disso, foi criado alguns arquivos .h para tornar o código mais legível e organizado.

Como discutido anteriormente, nos arquivos `location_array.h` e `location_array.c` nós temos a implementação das estruturas de dados e funcionalidades do nosso sistema com relação às operações feitas com os postos de vacinação. Temos também os arquivos `common.h` e `common.c` que implementam algumas funcionalidades que são comuns tanto no código do cliente quanto no código do servidor, como por exemplo, a função de recepção de uma mensagem através de múltiplos pacotes.

4.2 "Sincronização" dos fluxos de operação

É importante notar que o fluxo de operação do cliente é ligeiramente diferente do fluxo de operação do servidor. Enquanto que no cliente nós temos um loop de envio e recepção de mensagens; no servidor nós temos o oposto, ou seja, a recepção, tratamento e envio de respostas.

Para que o código fique mais legível e para que o servidor seja implementado de maneira mais fácil, foram utilizadas diversas variáveis "flag" indicando se algum evento ocorreu durante a comunicação, como por exemplo o encerramento do servidor através do comando `kill`, a desconexão de um cliente dado uma violação de protocolo ou até mesmo a desconexão de um cliente por um motivo inesperado, como por exemplo quando o cliente executa o comando `ctrl C`.

4.3 Testes e verificações

Tanto o código do servidor quanto o do cliente foram testados em uma máquina local Linux versão Ubuntu 20.04 LTS. Foi desenvolvido testes manuais para os casos onde a mensagem ultrapassa o limite de bytes, para o caso onde o cliente informa uma localização fora do intervalo esperado ($0 \leq x, y \leq 9999$) e também para o caso onde o cliente desconecta através do comando `ctrl C` para testar se o servidor ainda mantém a sua execução de forma apropriada.

5 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. Por mais complexo que algumas partes tenham sido, o tempo fornecido foi suficiente para suprir todas as dúvidas e possibilitou uma aprendizagem mais profunda do paradigma cliente-servidor, bem como de outros conceitos da disciplina.