Documentação - Trabalho Prático 1

Disciplina: DCC023 - Redes de Computadores - TM
Aluno: Pedro Luís SIlva Giacomin
Matrícula: 2020021700

1. Introdução

Esse trabalho consiste na implementação de uma aplicação cliente-servidor em linguagem C usando a biblioteca POSIX. A aplicação consiste em um sistema de controle de uma casa inteligente.

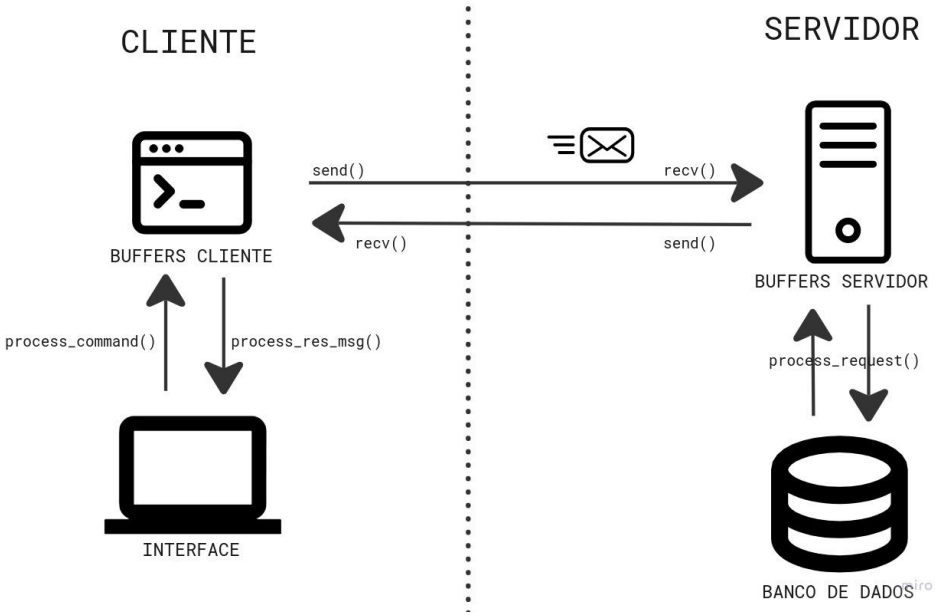
2. Arquitetura

A aplicação utiliza a arquitetura cliente-servidor, sendo que:

- **Cliente:** O cliente é quem solicita conexão com o servidor e envia mensagens de requisição para manipular os dispositivos da casa inteligente.
- **Servidor:** O servidor espera e aceita a conexão do cliente e executa as ações requeridas por ele, enviando mensagens de resposta para o cliente, sejam de controle ou de dados.
- **Banco de dados:** O servidor guarda os dispositivos da smart home e suas variáveis de estado em um banco de dados interno. Para manipular um dispositivo em um local (instalar, remover ou mudar o estado), o servidor altera as variáveis daquele dispositivo naquele local. Para consultar o estado (seja de um dispositivo ou de um local), o servidor apenas lê os valores das variáveis em seu banco de dados.
- **Mensagens:** Como especificado no protocolo, todas as mensagens são vetores de caracteres ASCII (strings em C). Tanto no cliente quanto no servidor, as mensagens em construção são armazenadas em um buffer desse tipo para posteriormente serem enviadas para o servidor (no caso de mensagens de requisição) ou para o cliente (no caso de mensagens de resposta).
- **Interface:** O cliente tem uma interface que é encarregada de receber os comandos do usuário no terminal a partir do teclado, e dispor informações da resposta do servidor ao usuário no terminal, que podem ser mensagens de sucesso, falha ou de dados.

As mensagens são de 9 tipos possíveis: INS_REQ, REM_REQ, CH_REQ, DEV_REQ, LOC_REQ, DEV_RES, LOC_RES, ERROR e OK, por isso para elas foi definido o enum MSG_TYPE. Para facilitar a manipulação das mensagens, foi definida a função `parse_msg_type`, que recebe uma string com o tipo da mensagem e retorna o enum correspondente. Tanto o enum quanto essa função foram implementados no arquivo ‘common.c’, já que são comuns ao cliente e ao servidor.

Abaixo são mostradas as entidades do sistema e as funções usadas em cada uma das fases, as quais serão detalhadas a seguir.



3. Cliente (implementação)

a. Conexão com o servidor

O IP e a porta à qual o cliente vai se conectar devem ser fornecidas na linha de comando e são armazenadas em uma variável do tipo [sockaddr], do POSIX. Esses endereços são então passados à função `socket()`, que inicializa o socket, e o socket é passado para a função `connect()`, que estabelece a conexão.

b. Troca de mensagens

Após a conexão com o servidor, o cliente entra em um loop infinito (a princípio) para receber comandos do teclado, enviar e receber mensagens do servidor e dispor na tela as respostas do servidor.

i) Recebe o comando:

- `char *buf` : buffer usado para guardar o comando recebido do teclado em formato de string.
- `fgets()`: função de <stdio.h> usada para receber do teclado o comando em formato de string e salvá-lo em buf.

ii) Trata o comando e cria a mensagem de requisição:

- `unsigned process_command(char *comando, char *msg_out)`: Essa função recebe a string de comando e constrói a string da mensagem de requisição (as terminadas em REQ) no formato definido pelo protocolo (especificado na descrição do TP). A mensagem de requisição resultante é salva em um buffer de `char*`, passado no segundo parâmetro da função.
  A lógica de implementação se baseia nas funções de manipulação de string:
  + `strtok()`: usada para cortar a string do comando em substrings, para coletar os dados inseridos pelo usuário.
  + `strcmp()`: usada para reconhecer as strings do comando e construir as mensagens de requisição de acordo com ele.

    + `strcpy()`: usada para copiar uma string para outra.
    + `strcat()`: usada para concatenar uma string a outra, permitindo contruir a mensagem passo a passo.

  A função faz testes ao longo de todo o tratamento do comando para averiguar se ele foi digitado corretamente e retorna:
    + 0, caso haja algum erro.
    + 1, caso não haja erro.
- `char *msg_buf` : buffer usado para guardar a mensagem de requisição que será enviada ao servidor, alocado dinamicamente com o tamanho de 500B (tamanho máximo permitido para a mensagem pelo protocolo)

iii) Envia mensagem de requisição e recebe mensagem de resposta:
- `send()` : função de POSIX usada para enviar a mensagem de requisição, salva em `msg_buf`, ao servidor a partir do socket definido anteriormente.
- `char *buf_res` : buffer usado para guardar a mensagem de resposta para ser tratada posteriormente, alocado dinamicamente com o tamanho de 500B.
- `recv()` : função de POSIX usada para receber a mensagem que chega no socket, e salvá-la no buffer definido acima.

iv) Trata a mensagem de resposta
- `void process_res_msg(char *msg_in, char *str_out, char *req_msg)`: Essa função recebe a mensagem de resposta e constrói o aviso que deve ser disposto no terminal para o usuário. O aviso pode ser de sucesso (OK), falha (ERROR) ou carregar o estado de um dispositivo ou local (DEV_RES ou LOC_RES).

  A função deve receber a mensagem de requisição que antecede a resposta para ser capaz de dispor na tela o local e o dispositivo (no caso da DEV_RES e da LOC_RES), pois o protocolo define que essas informações não estão presentes nessas mensagens que vêm do servidor.
- `char *warn` : buffer usado para guardar o aviso ou os dados do dispositivo recebidos da mensagem de resposta do servidor, para posteriormente ser didposto na tela, alocado dinamicamente com o tamanho de 1024B.

c. Considerações finais sobre o cliente

O cliente sai do loop infinito de troca de mensagens, caso o valor retornado pela função process_command seja 0 (acusando erro de digitação no comando), ou caso o comando recebido seja “kill”. Ao sair do loop infinito o cliente encerra a conexão, usando a função `close()` do POSIX.
Toda a memória alocada dinamicamente para os buffers é liberada ao final do loop.

4. Servidor (implementação)

a. Banco de dados

Foram definidas duas estruturas para viabilizar a manipulação dos dispositivos pelo servidor.
- struct dispositivo: todo dispositivo presente na **tabela I** possui
    + `unsigned id`
    + `unsigned ligado`
    + `unsigned dados`
- struct local: cada local possui

    + `unsigned id`
    + `struct dispositivo[5]`

A lógica do banco de dados é como segue:

- O banco de dados do servidor é simplesmente um vetor de `struct local` com 5 posições.
- A posição i do vetor é reservada ao local de id i + 1, segundo **tabela II**.
- Como já mostrado, cada local tem em sua estrutura um vetor de 5 dispositivos, a posição j do vetor é reservada ao dispositivo de id j + 1, segundo **tabela I**.
- Quando o programa servidor é executado, o banco de dados é inicializado da seguinte maneira:
    + O id de cada local é inicializado com seu valor segundo tabela II.
    + Todos os parâmetros, inclusive o id, de todos os dispositivos de cada local são inicializados com 0.
- O que indica se um dispositivo não está instalado é se seu id tem valor 0.
- Essa lógica não permite ter 2 dispositivos iguais instalados no mesmo local.

b. Troca de mensagens

i.) Recebimento da mensagem de requisição

- `char *req_msg` : buffer que guarda a mensagem de requisição que chega no socket do servidor, alocado estaticamente com tamanho 500B.
- `recv()` : função de POSIX usada para receber a mensagem que chega no socket, e salvá-la no buffer definido acima.

ii) Processamento da requisição

- `void process_request(char *request, struct local locais[], char *response)`: Essa função atua diretamente no banco de dados do servidor, recebendo-o como parâmetro. Faz as alterações ou consulta definida na mensagem de requisição e salva a mensagem de resposta ao cliente na string response, cujo endereço é recebido no terceiro parâmetro da função.

  Para acessar um dispositivo <dev_id> em um local <loc_id>, acessa-se o vetor locais[] como a seguir `locais[loc_id - 1].dispositivos[dev_id - 1]`.

  Para conseguir os dados da string da mensagem de requisição, a função utiliza o `strtok()`, explicado anteriormente, e usa a função `atoi()` para transformar os dados do tipo string para inteiro, viabilizando salvá-los no banco de dados:
    + instalar dispositivo: acessa o dispositivo como mostrado acima e altera o id e os dados.
    + remover dispositivo: acessa o dispositivo como mostrado acima e zera todos os seus atributos (id, ligado e dado).
    + alterar estado de dispositivo: acessa o dispositivo como mostrado acima e altera seus atributos `ligado` e `dado`.
    + mostrar estado de dispositivo: acessa o dispositivo como acima e consulta seus atributos `ligado` e `dado`.
    + mostrar estado do local: acessa o local `locais[loc_id - 1]` e consulta os atributos `ligado` e `dado` de todos os seus dispositivos que tenham id diferente de 0, por meio de um loop for.

As mensagens de resposta DEV_RES e LOC_RES são construídas no interior dessa função, no tratamento dos casos DEV_REQ e LOC_REQ.

Para identificar falhas e são realizados testes para cada uma das 4 situações de erro possíveis:

+ ERROR 01 - dispositivo não instalado: checa se o id do dispositivo na posição designada para ele é 0.
+ ERROR 02 - nenhum dispositivo instalado: só acontece para a funcionalidade de mostrar estado do local; usa uma variável count inicializada com 0 e itera ela toda vez que um dispositivo tem id diferente de 0 no local que está sendo consultado; testa ao final de todas as iterações do loop se count permanece em 0.
+ ERROR 03 - id de dispositivo inválido: usa a função auxiliar unsigned `is_dev_id_valid(int dev_id)` que checa se o id está entre `MIN_DEV_ID` e `MAX_DEV_ID` (nesse caso 1 e 5).
+ ERROR 04 - id de local inválido: usa a função auxiliar unsigned `is_loc_id_valid(int loc_id)` que checa se o id está entre `MIN_LOC_ID` e `MAX_LOC_ID` (nesse caso 1 e 5).

As mensagens de OK e ERROR são construídas por funções auxiliares explicadas abaixo

- `void build_error_msg(char *msg_out, unsigned codigo)`: recebe um ponteiro para a string onde a mensagem vai ser salva e o código da mensagem de erro; aloca a string dinamicamente.
- `void build_ok_msg(char *msg_out, unsigned codigo)`: recebe um ponteiro para a string onde a mensagem vai ser salva e o código da mensagem de confirmação; aloca a string dinamicamente.

iii) Envia mensagem de resposta:

- `char *res_msg` : buffer usado para guardar a mensagem de resposta para ser enviada, alocado dinamicamente com o tamanho de 500B (e liberada depois de enviada)
- `send()` : função de POSIX usada para enviar a mensagem de resposta, salva no buffer acima, ao cliente a partir do socket definido anteriormente.

5. Discussão e Conclusão

A aplicação atende a todos os requisitos especificados na descrição do TP1 e implementa o protocolo corretamente. A lógica de banco de dados, porém, pode passar por algumas melhorias, como:

- suportar dois dispositivos iguais instalados no mesmo local
- limitar os valores possíveis para os estados de cada dispositivo (alguns têm intervalos bem definidos, como a câmera que pode ir de 0 a 360 graus)

Alguns desafios foram encontrados ao longo da implementação. Dentre eles estão:

- Alguns problemas com alocação dinâmica para as strings, principalmente com a função `realloc()`, por isso alguns buffers são alocados dinamicamente e outros estaticamente. Como todas as strings a serem trocadas entre o cliente e o servidor ou entre a aplicação e o usuário são curtas, não houve problema alocá-las estaticamente. Para isso foram definidas duas constantes, MSGSZ e BUFSZ, respectivamente com 500B e 1024B.
- A princípio tentei usar uma `struct req_msg` ipara guardar as mensagens de requisição. Na primeira versão do projeto ela era usada como uma struct intermediária entre a

mensagem de o comando recebido do teclado (string) e a mensagem enviada ao servidor (string). Porém alguns problemas com alocação dinâmica dessas strings e as particularidades de cada mensagem tornaram inviável o uso dessa struct intermediária, por isso optei por transformar o comando diretamente em uma mensagem de requisição por meio da função `process_command`. Como há poucos tipos de mensagens e cada uma carrega poucos dados, isso foi viável, mas caso contrário, o uso de uma struct intermediária provavelmente ajudaria.

Por fim, a aplicação utiliza o POSIX como mostrado nas aulas fornecidas como material de apoio e funciona sobre o protocolo TCP, além de suportar as versões 4 e 6 do protocolo IP.