Documentação TP2

Introdução

Este é o segundo Trabalho Prático da disciplina Redes de Computadores, cujo objetivo é exercitar os conceitos aprendidos na programação em redes, sendo que nesse caso foi usado um modelo de comunicação Cliente-Servidor - com 2 servidores e múltiplos clientes
O contexto do trabalho são
campi inteligentes conectados à sistemas de geração de energia e de iluminação. O código fonte desenvolvido conta com 3 arquivos de implementação com nomes auto-explicativos:

- server.c
- client.c
- commom.c - com funções comuns ao cliente e servidor.

É valido destacar que o código fonte foi construído tendo como base a playlist Introdução à Programação em Redes, do Professor Ítalo Cunha, dando ênfase à última aula da playlist, que trata do uso de **threads** para comunicação multi-clientes. Para fins de simplificação, o código do Prof. Ítalo não será discutido.

Arquitetura

Descrição em alto nível

Toda o esqueleto do estabelecimento da conexão entre o servidor e o cliente já havia sido desenvolvida na playlist de programação em redes, de modo que não foi necessário a preocupação em atos como definição de portas, protocolos, instanciação de sockets, estabelecimento de conexão, envio de mensagens, tratamento de erros referentes à conexão e criação de threads. Entretanto, para boa realização do tema proposto, foi necessário refinar os conceitos que já estavam prontos a fim de adequá-los às necessidade do trabalho.

O servidor

São dois os servidores deste trabalho: o servidor na Subestação de Energia (Servidor SE) e o servidor do Sistema de Controle de Iluminação Inteligente (Servidor SCII). Ambos os servidores são inicializados e aguardam a conexão de clientes. Esses servidores sempre têm, durante todo o tempo de execução, os mesmos clientes - isto é, um cliente não deve conectar-se à apenas um servidor. A função deles é, conforme solicitação do cliente, informar acerca da geração e consumo de energia, informações essas que estão armazenadas nos servidores.

As solicitações tradas por esses servidores são:

Mensagem	Descrição
display info se	Informa a produção elétrica
display info scii	Informa o consumo elétrico
query condition	Informa sobre a produção e faz ajutes necessários no consumo
kill	Finaliza a conexão com o cliente

É válido ressaltar que os servidores aceitam cada, no máximo, 10 conexões simultânes, atribuem IDs únicos a cada cliente e usam threads para administrar essas conexões paralelamente.

O cliente

O cliente tem funcionalidade muito simples: ao abrir a conexão, ele faz requisições ao servidor e exibe na tela as informações pertinentes. É válido destacar que, ao conectar-se, o cliente primeiro recebe uma mensagem de boas vindas, e então entra em um loop while de envio e recebimento de mensagens com os servidores (send, recv).

server.c

Um único arquivo.c é capaz de lidar com os tipos de servidores, o SE e o SCII. Foi decido que o server.c contaria com duas implementações de threads, e conforme argumentos passado no executável seria chamada a thread correta:

```
pthread_t tid;
if (atoi(argv[2]) == 12345) {
pthread_create(&tid, NULL, client_thread_SE, cdata);
pthread_detach(tid);
}
if (atoi(argv[2]) == 54321) {
pthread_create(&tid, NULL, client_thread_SCII, cdata);
pthread_detach(tid);
}
```

É válido destacar que os dados são passados para a thread através de uma struck chamada client_data que contém o socket do cliente, storage e ID. Além disso, os dados de produção e consumo armazenados nos servidores são, na verdade, variáveis globais podendo ser lidas e alteradas pelas threads.

O código *main()* é relativamente simples. Para além do processo de conexão, roda-se um loop while que se encarrega de gerar os valores aleatorios de producao e consumo, atribui IDs sequenciais a cada novo cliente e seleciona a devida thread conforme argumento passado:

```
// Bloqueia o mutex antes de acessar e modificar a contagem de clientes
pthread_mutex_lock(&client_count_mutex);
if (client_count >= MAX_CLIENTS) {
    const char *response = "Client limit exceeded\n";
    send(csock, response, strlen(response)+1, 0);
    close(csock);
    printf("Client limit exceeded\n");
} else {
    client_count++;
    struct client_data *cdata = malloc(sizeof(*cdata));
    if (!cdata) {
        logexit("malloc");
    }
    cdata->csock = csock;
    memcpy(&(cdata->storage), &cstorage, sizeof(cstorage));

    // Atribui um ID único ao cliente
    pthread_mutex_lock(&client_id_mutex);
    cdata->client_id = next_client_id++;
    pthread_mutex_unlock(&client_id_mutex);

    printf("Client %d added\n", cdata->client_id);
```

Fig 1 - Atribuilão de IDs

client_thread_SE()

A thread do Servidor SE, após ser iniciada, roda um loop while para recebimento de mensagens. São usadas estruturas condicionais para tratar as requisiçoes mostradas na Tabela no início deste documento. Caso a requisição não esteja na tabela, o servidor apenas confirma ao cliente que recebeu sua mensagem.

```c
char response[BUFSZ];
memset(response, 0, BUFSZ);
if(strcmp(buf, "kill\n") == 0 ) {
    printf("Servidor SE Client %d removed\n", cdata->client_id);
    memset(response, 0, BUFSZ);
    snprintf(response, BUFSZ, "Successful disconnect");
    send(cdata->csock, response, strlen(response)+1, 0);
    break;
} else if(strcmp(buf, "display info se\n") == 0 ) {
    printf("REQ_INFOSE\n");
    printf("REQ_INFOSE %d\n", producao);
    memset(response, 0, BUFSZ);
    snprintf(response, BUFSZ, "producao atual: %d kWh", producao);
    send(cdata->csock, response, strlen(response)+1, 0);

} else if(strcmp(buf, "query condition\n") == 0) {
    printf("REQ_STATUS\n");
    if(producao >= 41) {
        producao = geraProducaoSE(); // Gera novo valor aleatorio de producao

        memset(response, 0, BUFSZ);
        snprintf(response, BUFSZ, "estado atual: alta");
        send(cdata->csock, response, strlen(response)+1, 0);
    } else if(producao >= 31){
        producao = geraProducaoSE(); // Gera novo valor aleatorio de producao

        memset(response, 0, BUFSZ);
        snprintf(response, BUFSZ, "estado atual: moderada");
        send(cdata->csock, response, strlen(response)+1, 0);
    } else {
        producao = geraProducaoSE(); // Gera novo valor aleatorio de producao

        memset(response, 0, BUFSZ);
        snprintf(response, BUFSZ, "estado atual: baixa");
        send(cdata->csock, response, strlen(response)+1, 0);
    }
```

Fig 2 - Condicionais para tratamento das requisições no Servidor SE

Conforme pode ser visto, o servidor realiza as ações solicitadas conforme mensagem recebida pelo cliente. É importante destacar que o servidor **mantém a conexão aberta** a todo momento com o cliente - a menos se for enviado o comando ‘kill’.

client_thread_SCII()

A thread do Servidor SCII, após ser iniciada, roda um loop while para recebimento de mensagens. São usadas estruturas condicionais para tratar as requisiçoes mostradas na Tabela no início deste documento. Caso a requisição não esteja na tabela, o servidor apenas confirma ao cliente que recebeu sua mensagem.

```
char response[BUFSZ];
memset(response, 0, BUFSZ);
if(strcmp(buf, "kill\n") == 0 ) {
    printf("Servidor SCII Client %d removed\n", cdata->client_id);

    memset(response, 0, BUFSZ);
    snprintf(response, BUFSZ, "Successful disconnect");
    send(cdata->csock, response, strlen(response)+1, 0);
    break;

} else if(strcmp(buf, "display info scii\n") == 0 ) {
    printf("REQ_INFOSCII\n");
    printf("REQ_INFOSCII %d%%\n", consumo);

    memset(response, 0, BUFSZ);
    snprintf(response, BUFSZ, "consumo atual: %d%% kWh", consumo);
    send(cdata->csock, response, strlen(response)+1, 0);

} else if(strcmp(buf, "REQ_UP") == 0){
    int old_consumo = consumo;
    int new_consumo = aumentaConsumoRandom(consumo);
    consumo = new_consumo;
    printf("RES_UP %d %d\n", old_consumo, new_consumo);

    memset(response, 0, BUFSZ);
    snprintf(response, BUFSZ, "RES_UP %d %d", old_consumo, new_consumo);
    send(cdata->csock, response, strlen(response)+1, 0);

} else if(strcmp(buf, "REQ_NONE") == 0) {
    puts(buf);

    memset(response, 0, BUFSZ);
    snprintf(response, BUFSZ, "RES_NONE %d ..", consumo ); // Envio de .. para controle
    send(cdata->csock, response, strlen(response)+1, 0);

} else if(strcmp(buf, "REQ_DOWN") == 0) {
    int old_consumo = consumo;
```

Fig 3 - Condicionais para tratamento das requisições no Servidor SCII

Conforme pode ser visto, o Servidor SCII é análogo ao Servidor SE.

client.c

Antes da explicação do código do Cliente propriamente dito, note que ele possue 2 funções para tratamento de buffers de caracteres. Essas duas funções são úteis para tratar as respostas do servidor:

```
/**
 * @brief Obtem a primeira palavra de um buffer de caracteres e escreve em outro b
 *
 * @param buffer  buffer de caracteres original
 * @param primeira buffer a ser escrito
 */
void primeira_palavra(const char *buffer, char *primeira);

/**
 * @brief Obtem a segunda e a terceira palavra de um buffer de caracteres e escrev
 *
 * @param buffer buffer de caracteres original
 * @param segunda  buffer 1 a ser escrito
 * @param terceira buffer 2 a ser escrito
 */
void segunda_terceira_palavra(const char *buffer, char *segunda, char *terceira);
```

Diferentemente do Servidor, o Cliente não faz uso de threads. Apesar de estar conectado simultaneamente a dois servidores, a comunicação por parte do cliente é feita de maneira sequencial. Isto é, ele estabelece uma conexão por vez, envia mensagens a um servidor por vez e trata as repostas uma por vez. Veja nas fotos abaixo que a Conexao com Servidor SE é feita a partir da linha 93, e a Conexao

com Servidor SCII é feita a partir da linha 124. Isso quer dizer que o Cliente, efetivamente, se conecta primeiro com o Servidor SE e recebe seu ID, e **em sequência** se conecta com o Servidor SCII e recebe seu ID.

```
93      /* ---------- Conexao com Servidor SE ---------- */
94      bool close_SE = false;
95      struct sockaddr_storage storage_SE;
96      if (0 != addrparse(argv[1], argv[2], &storage_SE)) {
97          usage(argc, argv);
98      }
99
100     int s_SE;
101     s_SE = socket(storage_SE.ss_family, SOCK_STREAM, 0);
102     if (s_SE == -1) {
103         logexit("socket");
104     }
105     struct sockaddr *addr_SE = (struct sockaddr *)(&storage_SE);
106     if (0 != connect(s_SE, addr_SE, sizeof(storage_SE))) {
107         logexit("connect");
108     }
```

Fig 4 - Conexao com Servidor SE, estabelecida primeiro.

```
124     /* ---------- Conexao com Servidor SCII ----------*/
125     bool close_SCII = false;
126     struct sockaddr_storage storage_SCII;
127     if (0 != addrparse(argv[1], argv[3], &storage_SCII)) {
128         usage(argc, argv);
129     }
130
131     int s_SCII;
132     s_SCII = socket(storage_SCII.ss_family, SOCK_STREAM, 0);
133     if (s_SCII == -1) {
134         logexit("socket");
135     }
136     struct sockaddr *addr_SCII = (struct sockaddr *)(&storage_SC
137     if (0 != connect(s_SCII, addr_SCII, sizeof(storage_SCII))) {
138         logexit("connect");
139     }
```

Fig 4 - Conexao com Servidor SE, estabelecida em sequência.

Após estabelecer as conexões e receber os IDs, o arquivo roda um loop while no qual é possível mandar uma mensagem ao servidor atraves de input via teclado, e então são tradadas as respostas do servidor através de estruturas condicionais (assim como é feito no Servidor).

```
155     while (1) {
156         if(!mensagemDummy) {
157             // Input mensagem
158             char buf[BUFSZ];
159             memset(buf, 0, BUFSZ);
160             printf("mensagem> ");
161             fgets(buf, BUFSZ - 1, stdin);
162             size_t msg_len = strlen(buf);
163
164             // Enviando mensagem para servidor SE
165             if (send(s_SE, buf, msg_len, 0) != msg_len) {
166                 logexit("send");
167             }
168             // Enviando mensagem para servidor SCII
169             if (send(s_SCII, buf, msg_len, 0) != msg_len) {
170                 logexit("send");
171             }
172         } else {
173             send(s_SE, "...", strlen("...")+1, 0);
174             send(s_SCII, "...", strlen("...")+1, 0);
175         }
176         mensagemDummy = false;
```

Fig 5 - Envio de mensagem. Note que o envio é sequencial, não paralelo

> 💡 A mensagemDummy é um booleano que indica se deve-se receber um input do teclado para a próxima mensagem ou simplesmente enviar um dummy para o servidor. Esse dummy é enviado após as solicitações query condition afim de alinhar o send() e recv() de Cliente e Servidor.

Veja abaixo as estruturas condicionais para tratamento das respostas recebida do Servidor SE e do Servidor SCII:

```
178         // Recebendo resposta servidor SE
179         count_SE = recv(s_SE, buf_SE, BUFSZ - 1, 0);
180         if (count_SE < 0) {
181             logexit("recv");
182         } else if (count_SE == 0) {
183             printf("Connection closed by server\n");
184             break;
185         }
186 
187         if (strcmp(buf_SE, "Successful disconnect") == 0) {
188             printf("Successful disconnect\n");
189             close_SE = true;
190         } else if (strcmp(buf_SE, "estado atual: alta") == 0)
191             mensagemDummy = true;
192             puts(buf_SE);
193             send(s_SCII, "REQ_UP", strlen("REQ_UP"), 0);
194         } else if (strcmp(buf_SE, "estado atual: moderada") =
195             mensagemDummy = true;
196             puts(buf_SE);
197             send(s_SCII, "REQ_NONE", strlen("REQ_NONE"), 0);
198         }else if (strcmp(buf_SE, "estado atual: baixa") == 0)
199             mensagemDummy = true;
200             puts(buf_SE);
201             send(s_SCII, "REQ_DOWN", strlen("REQ_DOWN"), 0);
202         }else {
203             puts(buf_SE);
204         }
```

Fig 6 - Condicionais p/ respostas do SE

```
207         // Recebendo resposta servidor SCII
208         count_SCII = recv(s_SCII, buf_SCII, BUFSZ - 1, 0);
209         if (count_SCII < 0) {
210             logexit("recv");
211         } else if (count_SCII == 0) {
212             printf("Connection closed by server\n");
213             break;
214         }
215 
216         char action[50];
217         primeira_palavra(buf_SCII, action);
218 
219         if (strcmp(action, "RES_UP") == 0) {
220             memset(action, 0, BUFSZ);
221             char old_value[5];
222             char new_value[5];
223             segunda_terceira_palavra(buf_SCII, old_value, new
224             printf("consumo antigo: %s\nconsumo atual: %s\n",
225 
226         } else if(strcmp(action, "RES_NONE") == 0){
227             memset(action, 0, BUFSZ);
228             char old_value[5];
229             char dummy[5];
230             segunda_terceira_palavra(buf_SCII, old_value, dum
231             printf("consumo antigo: %s\n", old_value);
232 
233         }else if(strcmp(action, "RES_DOWN") == 0) {
234             memset(action, 0, BUFSZ);
235             char old_value[5];
236             char new_value[5];
237             segunda_terceira_palavra(buf_SCII, old_value, new
238             printf("consumo antigo: %s\nconsumo atual: %s\n",
239 
240         }  else if (strcmp(buf_SCII, "Successful disconnect")
241             printf("Successful disconnect\n");
242             close_SCII = true;
```

Fig 6 - Condicionais p/ respostas do SCII

Por fim, caso o Cliente recebe a mensagem “Successful disconnect” de qualquer servidor, a conexão é fechada.