

Trabalho Prático de Redes - Sistema de Monitoramento de Falhas em Redes Elétricas

João Lucas Simões Moreira, 2022043485

1. Introdução

O presente trabalho prático teve como objetivo desenvolver uma rede que possuísse dois servidores com papéis distintos — um Servidor de Localização (SL) e um Servidor de Status (SS) — e capacidade para gerenciar até 15 sensores simultaneamente, de modo a comporem um sistema de monitoramento de falhas em redes elétricas.

As responsabilidades dos servidores são divididas: o SL gerencia a localização de cada sensor (associada a um número de 1 a 10), enquanto o SS monitora o status reportado por eles (0 para "sem risco" e 1 para "risco detectado"). Uma pane elétrica é efetivamente detectada e alertada pelo sistema quando um sensor reportar o status 1.

O protocolo de comunicação foi construído utilizando a biblioteca *sockets* em C. O primeiro servidor que entrasse na rede teria dois *sockets* para a comunicação com outros servidores e com o peer, e o segundo teria apenas o *socket* de comunicação com o peer. Além destes, ambos teriam um *socket* para ouvir por novos sensores, e cada sensor que entrasse na rede seria atribuído à um *socket*. Do lado dos sensores, cada um teria dois *sockets*, um para cada servidor.

2. Mensagens

As mensagens são fundamentais para o estabelecimento do protocolo adotado no trabalho. Todas possuem até 500 bytes e, no total, existem 16 tipos distintos que podem ser trocadas ao longo da execução deste sistema:

1. 6 mensagens de controle, que são:
 - a. Requisição e resposta das conexões entre peers;
 - b. Requisição e resposta das conexões entre servidores e sensores;
 - c. Requisições de desconexão dos peers e dos sensores.
2. 2 mensagens de erro/confirmação;
3. 8 mensagens de dados, que são as principais mensagens trocadas pelos componentes do sistema. São responsáveis por trocar status, localizações e áreas.

Elas são enviadas e recebidas por meio das funções *write()* e *read()*, respectivamente, que são adequadas para a comunicação entre *sockets* que utilizem o protocolo TCP. Em relação ao formato das mensagens, foi construída uma *struct* em C, denominada *message*, que possui um inteiro tipo (de acordo com a tabela de mensagens presente nas especificações do trabalho) e uma string que representa o payload. As imagens abaixo representam cada detalhe discutido neste parágrafo.

<pre>typedef struct { int type; char payload[MAX_MSG_SIZE]; } message;</pre>	<pre>// message ReceiveRawMessage(int socket_fd) { message msg; ssize_t n = read(socket_fd, &msg, sizeof(msg)); if (n < 0) error("ERROR reading from socket"); return msg; }</pre>	<pre>void SendMessage(int type, char* payload, int socket_fd) { message msg; memset(msg.payload, MAX_MSG_SIZE); msg.type = type; strncpy(msg.payload, payload, MAX_MSG_SIZE - 1); msg.payload[MAX_MSG_SIZE - 1] = '\0'; int n = write(socket_fd, &msg, sizeof(msg)); if (n < 0) error("ERROR writing to socket"); }</pre>
Figura 1: estrutura das mensagens no trabalho prático, onde MAX_MSG_SIZE = 500.	Figura 2: função responsável por receber mensagens.	Figura 3: função responsável por enviar mensagens.

Por fim, é válido ressaltar que toda requisição e resposta enviados no sistema são anotadas pela seguinte mensagem, exibida no terminal: "Sending <Requisição/ Resposta> to <Destino>".

3. Arquitetura

Para a arquitetura, foi adotado um P2P para construir a comunicação entre os servidores SS e SL, e um cliente-servidor entre cada um destes e os sensores conectados na rede. Um esquema pode ser visto na imagem abaixo.

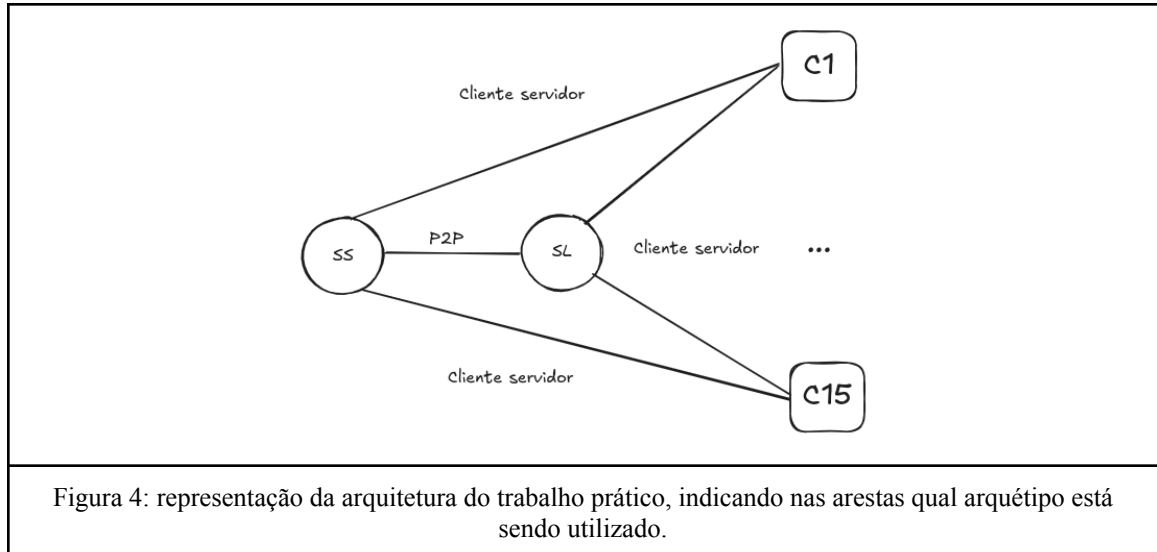


Figura 4: representação da arquitetura do trabalho prático, indicando nas arestas qual arquétipo está sendo utilizado.

Na conexão P2P, conforme visto anteriormente neste relatório, os servidores teriam um canal de comunicação direta por meio de um *socket*. No entanto, somente um deles terá um *socket* para ouvir por novos peers. Isso se deve ao fato de que é possível dar apenas um *bind* por vez na porta 64000, adotada como padrão para esse tipo de conexão neste projeto. Se ambos os peers conectados dessem *bind*, um deles teria sua execução interrompida.

Para lidar com entradas que podem surgir de múltiplas fontes, foi utilizada a função *select()* da biblioteca de *sockets* que monitora:

- No servidor: mensagens do peer conectado, requisições de conexão de outros servidores, requisições de conexão de novos clientes, mensagem dos clientes já aceitos na rede e comandos enviados pelo terminal;
- No sensor: mensagens dos servidores conectados e comandos enviados pelo terminal.

Cada um desses elementos citados anteriormente é mapeado para um *file descriptor* representado pelos *sockets* e pela entrada padrão *STDIN_FILENO*. O loop principal dos códigos do sensor e do servidor é responsável por monitorar estas diferentes fontes utilizando as primitivas *FD_SET* e *FD_ISSET*.

4. Servidor

Em relação aos servidores, como foi construído somente um código em C, foi necessário estabelecer padrões para determinar quem seria o SL e quem seria o SS. Deste modo, o servidor que se inicializar com a porta 60000 será o SL; com a 61000, o SS. Por fim, ambos se conectam na porta 64000 para se comunicarem de modo P2P. Para iniciarem suas execuções, basta digitar `./server <IP> <Porta P2P> <Porta que escuta por clientes>` no terminal.

O seguinte fluxo foi adotado para estabelecer as comunicações entre servidores:

- O primeiro servidor que se estabelece tenta se conectar a algum outro que existe na rede, utilizando a função *connect()*. Esta conexão falhará, pois esse foi o primeiro a entrar. Assim, o servidor estabelece um *socket* que escuta por novos servidores por meio de um *bind()* e um *listen()*;

- O segundo tenta se conectar e consegue, pois já existe um outro servidor o aguardando. Neste momento, é enviada uma mensagem do tipo *REQ_CONNPEER*, sem nenhum payload associado. Ao receber esta mensagem, o primeiro gera um ID e imprime no terminal “Peer <ID gerado> connected”, e envia este ID por meio da mensagem *RES_CONNPEER* ao servidor que se conectou;
- Ao receber *RES_CONNPEER*, o segundo servidor imprime “New Peer ID: <ID gerado>”, gera um ID ao primeiro, imprime no terminal “Peer <ID gerado> connected” e envia esse por meio da mensagem *RES_CONNPEER*;
- Quando o primeiro recebe seu ID, ele imprime “New Peer ID: <ID gerado>”, e isso efetivamente estabelece a comunicação entre servidores.

Após estabelecer comunicações, os servidores só voltam a se comunicar quando um destes sair da rede, ou quando um sensor executar o comando ***Check failure***, que será detalhado mais adiante neste relatório. Além disso, como dito anteriormente, é mantido aberto um socket que continua escutando por conexões novas de peers, apenas para rejeitá-las, pois podem haver no máximo 2 servidores conectados (ou seja, cada servidor tem no máximo 1 servidor conectado), e qualquer outro que tentar entrar na rede exibe a mensagem “Peer limit exceeded” e tem sua execução interrompida.

Para se comunicarem com os sensores, foi construído um *socket* que possuía a responsabilidade de ouvir por novas conexões. Sempre que uma fosse detectada, gerava-se um *socket* estabelecido que ficaria responsável por trocar mensagens entre os servidores e o novo sensor. Além disso, cada sensor foi armazenado em um vetor da estrutura abaixo, que contém todos os detalhes necessários para garantir o funcionamento ideal do sistema.

```
typedef struct {
    int status;
    int location;
    int socket_fd;
    char id[ID_LEN + 1];
} sensor_info;
```

Figura 5: Estrutura de dados para o armazenamento das informações dos sensores. Ressalta-se que o armazenamento é especializado por servidor: o campo *location* não é utilizado pelo SS, assim como o campo *status* não é utilizado pelo SL.

Os servidores também podem receber um comando no terminal que indica o término de suas atividades: ***kill*** (OBS: considerei que ***kill*** e ***close connection*** eram iguais, logo implementei apenas o primeiro). Ao executar este comando:

- O servidor envia a mensagem *REQ_DISCPEER*, com seu ID no payload, ao peer conectado, e espera que este confirme sua desconexão;
- O peer, ao receber esta mensagem, verifica se o ID recebido é o mesmo que está registrado. Se sim, exibe “Peer <ID do peer> disconnected” e envia uma mensagem *OK_SUCCESSFUL_DISCONNECT* ao seu peer, marcando a última mensagem trocada entre eles; Se não for, envia o erro *ERROR_PEER_NOT_FOUND*;
- Quando o primeiro servidor recebe a mensagem do tipo *OK*, ele exibe “Successful disconnect”, seguido de “Peer <seu ID> disconnected” e encerra sua execução; se recebe uma mensagem de erro, continua a executar.

Ao finalizar a conexão, o servidor que restou na rede passa a procurar por novos peers, voltando ao primeiro fluxo detalhado nesta seção. É válido ressaltar que os servidores também podem digitar ***kill*** no

terminal enquanto esperam por novas conexões, e que todos os sensores conectados na rede serão desconectados se um dos servidores encerrar sua execução.

Sobre tratamento de erros:

- O erro *ERROR_PEER_LIMIT_EXCEEDED* é enviado para todo servidor que tentar se conectar e não houver espaço na rede;
- O erro *ERROR_PEER_NOT_FOUND* é enviado se um peer pedir pra se desconectar e o ID dele não for encontrado;
- Os erros *ERROR_SENSOR_NOT_FOUND*, *ERROR_SENSOR_LIMIT_EXCEEDED* e *ERROR_LOCATION_NOT_FOUND* serão detalhados mais adiante, mas estão presentes na comunicação dos servidores com os sensores.

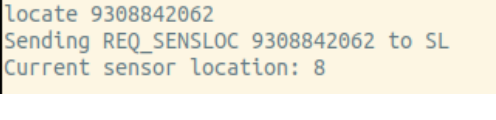
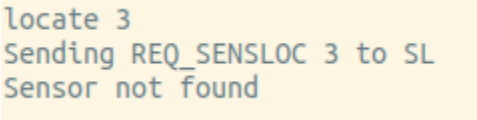
5. Sensores

Para se conectarem aos servidores da rede, os sensores devem inicializar sua com *./client <IP> <Porta SL> <Porta SS>*. Neste caso, também foi adotada a padronização citada na seção dos servidores, para facilitar as conexões. O fluxo da conexão de um sensor é o que segue:

- O sensor gera um ID para si mesmo e dá *connect()* nos dois servidores. Se for bem sucedido, envia a mensagem *REQ_CONNSEN* para ambos, cujo payload é o seu ID;
- Ao receber esta mensagem, o SL gera uma localização ao sensor e exibe em seu terminal “Client <ID sensor> added (Loc: <loc gerada>)”, e envia uma *RES_CONNSEN* com payload vazio ao sensor. O SS faz o mesmo, mas gera um status aleatório e exibe “Client <ID Sensor> added (Status: <status gerado>)”. Porém, caso a rede já esteja com os 15 sensores permitidos, é enviada a mensagem *ERROR_SENSOR_LIMIT_EXCEEDED* ao invés da usual;
- Quando o sensor recebe as *RES_CONNSEN*, ele exibe “SS/SL New ID: <ID gerado por ele>”, e em seguida exibe “OK(2)”. Caso ele receba o erro citado anteriormente, ele exibe “Sensor limit exceeded.” e encerra sua execução.

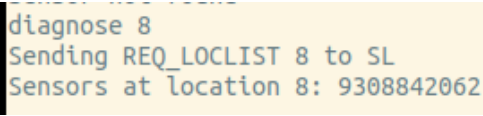
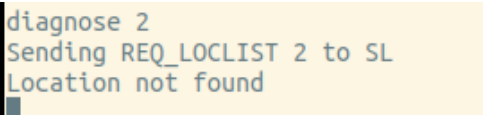
Todo sensor estará associado à uma localização e a um status, conforme dito anteriormente, e eles não terão estas informações armazenadas em sua memória. É válido ressaltar que nenhum sensor terá localização -1, pois esta é gerada aleatoriamente e corresponde a um valor entre 1 e 10. No total, podem ser executados até **quatro** comandos no terminal durante a execução:

- ***locate <ID>***: comando para buscar a localização de determinado sensor pelo seu ID. Ao executar este comando, o sensor envia a mensagem *REQ_SENSLOC* com o ID no payload ao SL, que procurará pelo ID informado. Caso este ID exista em sua base, o SL retorna a localização associada a ele por meio da mensagem *RES_SENSLOC*, e o sensor exibe “Current sensor location: <loc>”; caso contrário, é retornado um erro do tipo *ERROR_SENSOR_NOT_FOUND*, e o sensor exibe “Sensor not found”. Duas visualizações podem ser vistas abaixo:

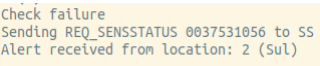
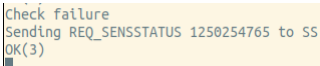
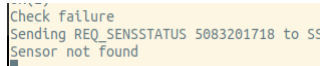
	
Figura 6: exemplo onde o <i>locate</i> funciona adequadamente.	Figura 7: exemplo onde o <i>locate</i> retorna um erro.

- ***diagnose <loc>***: comando para listar quais sensores estão em uma determinada localização. Ao executar este comando, o sensor envia a mensagem *REQ_LOCLIST* com a localização no payload ao SL, que busca todos os sensores presentes na localização informada e retorna uma *RES_LOCLIST* com esta lista no payload, e o sensor exibe “Sensors at location <loc>: <lista de

sensores>”; caso não hajam sensores, é retornado um *ERROR_LOCATION_NOT_FOUND*, e o sensor exibe “Location not found”. Duas visualizações podem ser vistas abaixo:

	
Figura 8: exemplo onde <i>diagnose</i> funciona adequadamente.	Figura 9: exemplo onde <i>diagnose</i> retorna um erro.

- **Check failure:** inicia uma verificação de status e localização de um sensor. Primeiro, o sensor envia seu ID ao SS via *REQ_SENSSTATUS* para que este verifique seu status. Se este for 0, o SS exibe “Sensor <ID> status = 0 (no failure detected)” e envia um *OK(3)*; caso seja 1, exibe “Sensor <ID> status = 1 (failure detected)” e envia ao SL uma mensagem *REQ_CHECKALERT* solicitando a área deste sensor. O SL a retorna com uma mensagem *RES_CHECKALERT*, ou devolve um erro *ERROR_SENSOR_NOT_FOUND* caso o sensor não seja encontrado na rede. Finalmente, o SS encaminha a área recebida para o sensor através de uma mensagem *RES_SENSSTATUS*, permitindo que ele exiba o alerta correto, ou devolve o *ERROR_SENSOR_NOT_FOUND*, onde o sensor exibirá a mesma mensagem no caso do comando *locate*. As três possíveis visualizações desse processo são mostradas abaixo:

		
Figura 10: exemplo onde o <i>check failure</i> funciona adequadamente.	Figura 11: exemplo onde o <i>check failure</i> retorna um <i>OK(3)</i> .	Figura 12: exemplo onde o <i>check failure</i> retorna um erro.

- **kill:** comando que encerra as comunicações entre sensor e servidores, e termina a execução do sensor. Ao executar, é enviada a mensagem *REQ_DISCSEN* com o ID no payload aos servidores, que removem o sensor de sua base e exibem “Client <ID do sensor> removed (Loc: <loc> | Status: <status>)”. Em seguida, é enviado um *OK(1)* ao sensor, que exibe “SS/SL Successful disconnect”.
 - OBS: considerei que *kill* e *close connection* eram iguais, logo implementei apenas o primeiro.

6. Discussão

O desenvolvimento do projeto apresentou desafios técnicos significativos, principalmente em relação à chamada de sistema *select()*. Esta foi fundamental para a arquitetura do servidor, pois permitiu o monitoramento simultâneo de múltiplas fontes de E/S — a entrada padrão para comandos via terminal, o socket de escuta para novos sensores, e os múltiplos sockets de dados para clientes e para a conexão P2P — sem recorrer a multithreading, o que simplificou o gerenciamento de estado e evitou complexidades de sincronização. A manutenção de um conjunto mestre de descritores de arquivo e a atualização constante do maior descritor foram cruciais para o funcionamento robusto do loop principal, garantindo que o servidor permanecesse responsivo a todos os eventos da rede.

Um dos principais obstáculos encontrados durante a implementação foi a presença de inconsistências e ambiguidades na especificação do trabalho. A princípio, a ausência de detalhes cruciais sobre a execução dos sensores e a presença de mensagens que não faziam muito sentido para a execução do trabalho atrasaram o desenvolvimento, só podendo ser resumido após encontros com o monitor da disciplina. E mesmo com esse canal de comunicação ainda ocorreu uma ambiguidade em relação à abertura

do servidor P2P, que causou um retrabalho. Como pontos positivos, o monitor sempre esteve disposto em nos ajudar e corrigir erros da parte dele. Foram disponibilizados bons casos de teste, e sempre que foi necessário perguntar algo no Moodle ele respondeu adequadamente.

Além disso, é válido retomar todas as decisões de projeto que foram tomadas ao longo do ciclo de desenvolvimento. Assumiu-se que sempre será utilizada a porta 64000 para a conexão P2P e as portas 60000 e 61000 para a conexão cliente-servidor. O cliente pode especificá-las na ordem que quiser ao inicializar, mas o servidor deve sempre seguir a ordem <Porta P2P> <Porta cliente-servidor>. Ademais, os *sockets* P2P sempre terão um *multibind* na porta 64000, ou seja, é possível dar *bind* múltiplas vezes nesta, apesar de que nunca será possível que tenham dois servidores ‘bindados’ ao mesmo tempo. Isso foi feito pois estava ocorrendo um erro no *bind* no momento em que o primeiro servidor que entrou na rede desse *kill*, fazendo com que o segundo servidor precisasse ouvir por novos servidores.

Por fim, é válido ressaltar que tudo aquilo que não foi discutido aqui (principalmente em nível de código) está devidamente explicado pelos comentários e pela organização estrutural dos três arquivos **common.c**, **server.c** e **client.c**.

7. Conclusão

Este trabalho prático contribuiu para a implementação bem-sucedida de uma arquitetura de rede funcional para o monitoramento de falhas elétricas, cumprindo os principais requisitos da especificação. Foram desenvolvidos servidores concorrentes capazes de comunicação P2P e gerenciamento de múltiplos sensores. O projeto representou uma aplicação prática e valiosa dos conceitos teóricos de redes de computadores, especialmente na programação de sockets TCP, no design de um protocolo de aplicação e, de forma crucial, no uso de I/O multiplexado com *select()* para gerenciar múltiplas conexões e entradas de forma eficiente e sem o uso de threads.

Apesar do sucesso na entrega de um sistema funcional, o processo de desenvolvimento foi marcado por desafios significativos, principalmente devido a ambiguidades e inconsistências na especificação do projeto. A superação desses obstáculos exigiu uma análise crítica do protocolo, a tomada de decisões de implementação para preencher lacunas e uma comunicação proativa para buscar esclarecimentos, o que constituiu uma experiência de aprendizado importante, simulando um cenário de desenvolvimento real. Em geral, foi uma experiência positiva de colaboração e proatividade para correr atrás das dificuldades.