UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA

GUSTAVO IVAN MULLER - 00303598
LUCCA FRANKE KROEFF - 00334209
MARIA EDUARDA TONETO - 00323751
SOFIA DE MORAES SAUTER BRAGA - 00333496

Trabalho Prático Parte 1: Threads, Sincronização e Comunicação

Relatório apresentado como requisito parcial para a disciplina de Sistemas Operacionais II N.

Prof. Weverton Cordeiro

Porto Alegre
2024

Sumário

Como foi implementado cada subserviço

Subserviço de Descoberta (Discovery Subservice)

O subserviço de descoberta foi implementado utilizando sockets UDP para enviar e receber mensagens de broadcast na rede. O objetivo é identificar hosts que estejam executando o mesmo programa na mesma rede e gerenciar sua presença.

Host::discovery

1. Criação do Socket: Um socket UDP é criado e configurado para permitir o envio de broadcasts.
2. Loop de Descoberta: Enquanto o estado do host for HostState::Discovery, pacotes de broadcast contendo o hostname e outros dados do host são enviados periodicamente.
3. Encerramento: Quando o estado do host muda para Awaken ou Asleep, o socket é fechado.

Manager::discovery

1. Criação do Socket: Um socket UDP é criado e configurado para reutilizar o endereço local.
2. Binding do Socket: O socket é associado a um endereço IP e porta para escutar mensagens.
3. Loop de Recepção: O manager fica em um loop infinito recebendo pacotes de descoberta.
4. Processamento do Pacote: Os pacotes recebidos são processados para extrair o hostname, IP, e endereço MAC do host.
5. Adição do Host: O host descoberto é adicionado à lista de hosts gerenciados, presente na própria estrutura do Manager.

Assim, essas funções permitem que os hosts anunciem sua presença na rede e que o manager detecte novos hosts.

Subserviço de Monitoração (Monitoring Subservice)

O subserviço de monitoramento foi implementado utilizando sockets TCP para manter e verificar a conexão entre o manager e os hosts na rede. O objetivo é garantir que os estados dos Hosts seja consistente.

Host::monitoring

1. Criação do Socket: Um socket TCP é criado e configurado para reutilizar o endereço local.
2. Binding do Socket: O socket é associado a um endereço IP e porta para escutar conexões.
3. Aceitação da Conexão: O host espera e aceita uma conexão do manager.
4. Loop de Comunicação: O host entra em um loop onde recebe pacotes do manager.
    - Troca de Estado Inicial: Se o estado do host for HostState::Discovery, ele muda para HostState::Asleep.
    - Verifica se Host quer para de executar: Se o estado for HostState::Exit, o host envia um pacote de saída e encerra a conexão.
    - Verfica se existe algum comando de troca de estado: Se for MessageType::SleepServiceCommand, o comando recebido é processado para mudar o estado do host para HostState::Awaken ou HostState::Asleep.
    - Resposta do subserviço de Monitoramento: Se for MessageType::SleepServiceMonitoring, o host responde com seu estado atual.

Manager::monitoring

1. Loop de Monitoramento:
    - Lock da Lista de Hosts: A lista de hosts é travada para evitar alterações durante a atualização de status.
    - Verificação de Conexão: Tenta conectar aos hosts não conectados.
    - Envio de Pacote de Monitoramento: Envia pacotes de monitoramento.
    - Processamento de Resposta: Atualiza o estado do host ou marca para remoção.
2. Unlock da Lista de Hosts: Destrava a lista de hosts.
3. Remoção de Hosts: Remove hosts desconectados.
4. Atraso: Aguardapara reiniciar o loop.

Essas funções permitem que o manager monitore continuamente os hosts, mantendo controle sobre seu estado e enviando comandos conforme necessário.

Subserviço de Gerenciamento (Management Subservice)

O subserviço de gerenciamento (Management Subservice) tem a responsabilidade de manter uma lista das estações participantes e monitorar o status de cada uma (asleep ou awaken). Na primeira fase do trabalho prático, essa lista é mantida exclusivamente na estação manager.

Manutenção da Lista de Hosts:

- Estrutura da Lista: A lista de hosts é armazenada no manager e inclui informações como hostname, endereço IP, endereço MAC e estado atual (Asleep ou Awaken).
- Proteção da Lista (seção crítica): A lista de hosts é protegida por mutexes para garantir que não ocorra concorrência durante a leitura e escrita, mantendo a integridade dos dados.

Abaixo, é dado como os outros subserviços se relacionam com o subserviço de gerenciamento:

Descoberta & Gerenciamento:

- Recepção de Pacotes de Descoberta: O manager recebe pacotes de descoberta enviados pelos hosts.
- Processamento de Pacotes: Cada pacote recebido é processado para extrair informações como hostname, IP, e MAC.
- Adição de Novos Hosts: Se um novo host é descoberto, suas informações são adicionadas à lista de hosts gerenciados pelo manager.

Monitoramento & Gerenciamento:

- Verificação de Conexão: O manager verifica periodicamente a conexão com cada host da lista.
- Envio de Pacotes de Monitoramento: Pacotes são enviados para monitorar o estado dos hosts.
- Atualização de Estado: As respostas dos hosts são processadas para atualizar seu estado na lista.
- Remoção de Hosts Inativos: Hosts que não respondem ou sinalizam que saíram são removidos da lista.

Interface & Gerenciamento:

- Visualização dos Hosts: O subserviço de interface exibe a lista de hosts contida na estação Manager, sendo atualizada em tempo real para refletir mudanças no estado dos hosts.

Subserviço de Interface (Interface Subservice)

O subserviço de interface utiliza a biblioteca ncurses para exibir informações em uma interface gráfica de texto. Ele permite que tanto o host quanto o manager apresentem o estado atual e outras informações relevantes de forma visual.

Host::interface

1. Criação da Janela: Uma nova janela ncurses é criada para exibir o estado do host.
2. Loop de Exibição:
    - A janela é atualizada periodicamente para mostrar o estado atual do host.
    - Se o estado for Exit, o loop é encerrado.
3. Encerramento: A janela ncurses é destruída.

Manager::interface

1. Criação da Janela: Uma nova janela ncurses é criada para exibir informações sobre os hosts gerenciados.
2. Loop de Exibição:
    - A janela é atualizada continuamente para mostrar hostname, IP, MAC e estado dos hosts.
    - Um cabeçalho com os títulos das colunas é exibido no topo.
3. Encerramento: O loop continua até que o programa seja finalizado.

Em quais áreas do código foi necessário garantir sincronização no acesso a dados

Foi crucial em nosso trabalho utilizarmos primitivas de exclusão mútua, pois no contexto do trabalho temos um ambiente multitarefa onde diversos processos ou threads podem acessar recursos compartilhados simultaneamente, sendo assim necessário garantir que o acesso a esses recursos seja sincronizado para evitar condições de corrida e inconsistências nos dados.

Tanto em host.cpp quanto em manager.cpp, precisamos ao inicializar o serviço usar primitivas de mutex_lock e mutex_unlock nos seguintes locais:

```
pthread_mutex_lock(&this->mutex_ncurses);
   initscr();
   refresh();
   curs_set(false);
pthread_mutex_unlock(&this->mutex_ncurses);
```

Precisamos utilizar essa primitiva de exclusão neste trecho de código pois essas funções manipulam a interface de usuário e precisam ser executadas sem interrupções para garantir a integridade da interface, além disso essas chamadas são usadas para proteger a manipulação da interface ncurses, garantindo que as operações sejam realizadas de maneira segura e sem interferências de outras threads.

```
pthread_mutex_lock(&this->mutex_ncurses);
   endwin();
pthread_mutex_unlock(&this->mutex_ncurses);
```

Precisamos utilizar essa primitiva de exclusão neste trecho de código pois a chamada de endwin modifica o estado global da interface do terminal, e deve ser executada sem interferências para garantir a integridade da interface e evitar comportamentos indesejados, e assim o uso de mutex em torno da chamada endwin() assegura que a operação de finalizar a interface ncurses seja realizada de maneira segura e sem interrupções por outras threads.

```
pthread_mutex_lock(&this->mutex_change_state);
   if (this->state != HostState::Exit) {
       this->prev_state = this->state;
       this->state = new_state;
   }
   pthread_mutex_unlock(&this->mutex_change_state);
```

Precisamos utilizar essa primitiva de exclusão neste trecho de código pois esta seção crítica garante que a verificação e a atualização do estado ocorram como uma operação atômica, sem interferências de outras threads. Assim, o estado anterior é armazenado corretamente antes da mudança, preservando o histórico do estado e garantindo que as operações subsequentes possam confiar na precisão dos dados de estado.

```
pthread_mutex_lock(&h->mutex_ncurses);
   output = create_newwin(height, width, start_y, start_x);
pthread_mutex_unlock(&h->mutex_ncurses);
```

Esta operação é crítica porque envolve a criação de um recurso compartilhado nova janela, que deve ser manipulado de forma exclusiva pela thread atual até que seja completamente inicializado.

```
pthread_mutex_lock(&h->mutex_ncurses);
       wclear(output);
       wprintw(output, "Current host state: %s\n", string_from_state(h->state).data());
       wprintw(output, "Press EXIT to quit");
       wrefresh(output);
pthread_mutex_unlock(&h->mutex_ncurses);
```

Sem o uso do mutex, poderia ocorrer uma condição de corrida onde várias threads tentam modificar a janela simultaneamente. Isso poderia resultar em problemas como partes da interface não serem atualizadas corretamente ou falhas na execução do programa.

```
pthread_mutex_lock(&h->mutex_ncurses);
   destroy_win(output);
pthread_mutex_unlock(&h->mutex_ncurses);
```

Destrói a janela. Esta é uma operação crítica, pois libera recursos da interface.

```
pthread_mutex_lock(&h->mutex_ncurses);
   input = create_newwin(height, width, start_y, start_x);
   wtimeout(input, h->input_timeout);
   wprintw(input, "> ");
   wmove(input, 0, 3);
pthread_mutex_unlock(&h->mutex_ncurses);
```

Esta é uma operação crítica, pois inicializa um recurso da interface. A criação e configuração da janela input devem ser realizadas de maneira exclusiva para evitar interferências. O mutex garante que essas operações sejam executadas de forma ordenada e sem conflitos.

```
pthread_mutex_lock(&h->mutex_ncurses);
       char ch = wgetch(input);
       if (ch == '\n') {
           wclear(input);
           if (in == "EXIT") {
               h->switch_state(HostState::Exit);
           }
           in.clear();
           wprintw(input, "> ");
           wmove(input, 0, 3);
       } else if (ch >= 0) {
           in.push_back(ch);
       }
 pthread_mutex_unlock(&h->mutex_ncurses);
```

Sem o mutex, várias threads poderiam tentar ler ou manipular a entrada do usuário simultaneamente, e isso é crucial para manter a integridade e a consistência da interface ncurses.

```
pthread_mutex_lock(&hosts_mutex);
   if (!this->has_host(host.name)) {
       this->hosts.push_back(host);
   }
pthread_mutex_unlock(&hosts_mutex);
```

A lista de hosts deve ser acessada e modificada de forma exclusiva para evitar inconsistências. O mutex garante que a verificação e a inserção do host sejam realizadas de maneira ordenada e sem conflitos. Sem o mutex, várias threads poderiam tentar verificar e modificar a lista de hosts simultaneamente, levando a resultados imprevisíveis como dados corrompidos.

```
pthread_mutex_lock(&hosts_mutex);
   for (auto it = this->hosts.begin(); it != this->hosts.end();) {
       KnownHost &h = *it;
       if (h.name == host.name) {
           close(host.sockfd);
           this->hosts.erase(it);
       } else it++;
   }
pthread_mutex_unlock(&hosts_mutex);
```

A lista de hosts deve ser acessada e modificada de forma exclusiva para evitar inconsistências. O mutex garante que a verificação e a remoção do host sejam realizadas de maneira ordenada e sem conflitos.

```
pthread_mutex_lock(&hosts_mutex);
   for (KnownHost h: this->hosts) copy.push_back(h);
pthread_mutex_unlock(&hosts_mutex);
```

A lista de hosts deve ser acessada de forma exclusiva para garantir que a cópia dos dados seja consistente. O mutex garante que a lista não seja modificada por outras threads enquanto a cópia está em andamento. Sem o mutex, outras threads poderiam tentar modificar a lista de hosts simultaneamente durante a cópia.

```
// lock so no changes are made to host list during status update
       pthread_mutex_lock(&m->hosts_mutex);

       for (auto it = m->hosts.begin(); it != m->hosts.end(); it++) {
           KnownHost &host = *it;

           if (!host.connected) {
               int sockfd = socket(AF_INET, SOCK_STREAM, 0);
               if (sockfd < 0) {
                   perror("Manager (Monitoring): Error creating socket");
                   continue;
               }

               struct sockaddr_in guest_addr;
               memset(&guest_addr, 0, sizeof(guest_addr));
               guest_addr.sin_family = AF_INET;
               guest_addr.sin_port = htons(PORT_MONITORING);
               inet_aton(host.ip.c_str(), &guest_addr.sin_addr);

               if (connect(sockfd, (struct sockaddr *) &guest_addr, sizeof(guest_addr)) < 0) {
```

```
                perror("Manager (Monitoring): Error connecting to host");
                close(sockfd);
                continue;
            }

            // Update host state to connected
            host.connected = true;
            host.sockfd = sockfd;
        }

        Packet request = Packet(MessageType::SleepServiceMonitoring, 0, 0);
        send_tcp(request, host.sockfd, PORT_MONITORING, host.ip);

        Packet response = rec_packet_tcp(host.sockfd);

        if (response.get_type() == MessageType::SleepServiceExit) {
            // Handle host exit
            remove.push_back(*it);
        } else {
            std::string state = response.pop();
            host.state = state_from_string(state);
        }
    }

    pthread_mutex_unlock(&m->hosts_mutex);
```

Neste trecho de código, o mutex é utilizado para garantir que a lista de hosts não seja modificada enquanto está sendo atualizada. A lista deve ser acessada de forma exclusiva para garantir que a atualização dos estados e conexões dos hosts seja consistente. O mutex garante que a lista não seja modificada por outras threads enquanto está sendo atualizada, e sem o mutex outras threads poderiam tentar modificar a lista de hosts ao mesmo tempo durante a atualização.

```
if (m->has_host(send_cmd.second)) {
    // lock so no changes are made to host list during command send
    pthread_mutex_lock(&m->hosts_mutex);

    for (auto it = m->hosts.begin(); it != m->hosts.end(); it++) {
        KnownHost &host = *it;
        if (host.name == send_cmd.second) {
            if (host.connected) {
                Packet command = Packet(MessageType::SleepServiceCommand, 0, 0);
                command.push(std::to_string(send_cmd.first));
                send_tcp(command, host.sockfd, PORT_MONITORING, host.ip);
            }
            break;
        }
    }
    pthread_mutex_unlock(&m->hosts_mutex);
}
```

Neste trecho de código, o mutex é utilizado para garantir que a lista de hosts não seja modificada enquanto o comando send está sendo enviado a um host. A lista deve ser acessada de forma exclusiva para garantir que a operação de envio de comandos seja consistente. O mutex garante que a lista não seja modificada por outras threads enquanto o comando está sendo enviado, e sem o mutex outras threads poderiam tentar modificar a lista de hosts simultaneamente durante o envio do comando.

```
while (1) {
       pthread_mutex_lock(&m->mutex_ncurses);

       wclear(output);
       wmove(output, 0, 0);
       wprintw(output, "Hostname");

       wmove(output, 0, 17);
       wprintw(output, "Endereço IP");

       wmove(output, 0, 37);
       wprintw(output, "Endereço MAC");

       wmove(output, 0, 58);
       wprintw(output, "Status");

       wmove(output, 1, 0);
       for (int i = 0; i < 64; ++i) {
           wprintw(output, "-");
       }

       /* get copy of hosts, good enough for printing */
       std::vector<KnownHost> hosts_c = m->get_hosts();

       for (long unsigned int i = 0; i < hosts_c.size(); ++i) {
           auto host = hosts_c[i];
           wmove(output, i + 2, 0);
           wprintw(output, host.name.c_str());

           wmove(output, i + 2, 17);
           wprintw(output, host.ip.c_str());

           wmove(output, i + 2, 37);
           wprintw(output, host.mac.c_str());

           wmove(output, i + 2, 58);
           wprintw(output, string_from_state(host.state).c_str());
       }

       wrefresh(output);
       pthread_mutex_unlock(&m->mutex_ncurses);

       usleep(m->sleep_output);
   }
```

A interface gráfica (ncurses) deve ser acessada de forma exclusiva para garantir que a atualização da tela seja consistente.

```
pthread_mutex_lock(&m->mutex_ncurses);
   input = create_newwin(height, width, start_y, start_x);
   wtimeout(input, m->input_timeout);
   wprintw(input, "> ");
   wmove(input, 0, 2);
```

```
pthread_mutex_unlock(&m->mutex_ncurses);
```

As operações ncurses precisam ser realizadas de forma atômica para garantir que a interface gráfica permaneça consistente. O mutex assegura que a janela de entrada seja criada e manipulada sem interferências.

```
while (1) {
       pthread_mutex_lock(&m->mutex_ncurses);
       char ch = wgetch(input);
       if (ch == '\n') {
           /* check for wakeup command */
           std::pair<int, std::string> ret = m->check_input(in);
           if (!ret.second.empty()) {
               m->cmd.push_back(ret);
           }
           wclear(input);
           in.clear();
           wprintw(input, "> ");
           wmove(input, 0, 2);
       } else if (ch >= 0) {
           in.push_back(ch);
       }
       pthread_mutex_unlock(&m->mutex_ncurses);

       usleep(m->sleep_input);
   }
```

Neste trecho de código, o mutex é utilizado para garantir que as operações de entrada e manipulação da janela de entrada (input window) sejam realizadas de forma segura e sem interferência de outras threads.

Descrição das principais estruturas e funções que a equipe implementou

Arquivo host.h :

Este arquivo define a classe Host e suas funções associadas. A classe Host gerencia o estado de um participante em uma rede que pode ser descoberto, monitorado e controlado por um gerenciador. O arquivo também define algumas funções auxiliares e estruturas de dados.

Estruturas:

- HostState: Enumeração que define os possíveis estados de um participante:
  - Discovery: O participante está em estado de descoberta.
  - Asleep: O participante está dormindo.
  - Awaken: O participante está acordado.
  - Exit: O participante está em processo de saída.
- ManagerInfo: Estrutura que armazena informações sobre o manager que descobriu o participante:
  - ip: Endereço IP do manager.
  - mac: Endereço MAC do manager.
  - name: Nome do manager.

Atributos:

- state: Estado atual do participante.
- prev_state: Estado anterior do participante.
- sck_discovery: Socket para descoberta.
- sck_monitoring: Socket para monitoramento.
- m_info: Informações do gerenciador.
- t_discovery, t_monitoring, t_interface, t_input: Threads para diferentes tarefas.
- mutex_change_state, mutex_ncurses: Mutexes para garantir operações seguras.

Métodos:

- init(): Inicializa o host, cria as threads e gerencia o ciclo de vida do participante.
- exit_handler(int sn, siginfo_t* t, void* ctx): Manipulador de saída para alterar o estado do participante para Exit.
- switch_state(HostState new_state): Muda o estado do participante de forma segura.
- Threads:
  - discovery(void *ctx): Thread responsável por enviar pacotes de descoberta.
  - monitoring(void *ctx): Thread responsável por monitorar o estado do participante e responder a comandos do manager.

    - interface(void *ctx): Thread responsável por atualizar a interface ncurses.
    - input(void *ctx): Thread responsável por lidar com a entrada do usuário.

Funções Auxiliares:

- state_from_string(std::string state): Converte uma string em um estado HostState.
- string_from_state(int state): Converte um estado HostState em uma string.
- create_newwin(int height, int width, int starty, int startx): Cria uma nova janela ncurses.
- destroy_win(WINDOW *local_win): Destrói uma janela ncurses.

Arquivo manager.h :

Este arquivo define a classe Manager e suas funções associadas. A classe Manager gerencia uma lista de participantes conhecidos e controla suas operações.

Estruturas:

- KnownHost: Estrutura que armazena informações sobre um participante conhecido:
    - ip: Endereço IP do participante.
    - mac: Endereço MAC do participante.
    - name: Nome do participante.
    - state: Estado do participante.
    - connected: Indica se o socket do participante está conectado.
    - sockfd: Descritor do socket do participante.

Atributos:

- hosts: Lista de participantes conhecidos.
- cmd: Fila de comandos.
- sck_discovery: Socket para descoberta.
- sck_management: Socket para gerenciamento.
- t_discovery, t_monitoring, t_management, t_command, t_interface, t_input: Threads para diferentes tarefas.
- hosts_mutex, mutex_ncurses: Mutexes para garantir operações seguras.

Métodos:

- init(): Inicializa o manager, cria as threads e gerencia o ciclo de vida do manager.
- exit_handler(int sn, siginfo_t* t, void* ctx): Manipulador de saída para cancelar todas as threads e fechar os sockets.

- add_host(KnownHost host): Adiciona um participante à lista de participantes conhecidos de forma segura.
- remove_host(KnownHost host): Remove um participante da lista de participantes conhecidos de forma segura.
- get_hosts(): Retorna uma cópia da lista de participantes conhecidos.
- has_host(std::string name): Verifica se um participante com um determinado nome está na lista de participantes conhecidos.
- check_input(std::string input): Verifica e retorna um comando e o nome do participante correspondente a partir de uma string de entrada.
- Threads:
    - discovery(void *ctx): Thread responsável por descobrir participantes na rede.
    - monitoring(void *ctx): Thread responsável por monitorar os participantes.
    - management(void *ctx): Thread responsável por gerenciar as conexões com os participantes.
    - command(void *ctx): Thread responsável por processar comandos.
    - interface(void *ctx): Thread responsável por atualizar a interface ncurses.
    - input(void *ctx): Thread responsável por lidar com a entrada do usuário.

Arquivo message.h :

Este arquivo define a classe Packet e funções auxiliares para lidar com mensagens de rede.

Estruturas:

- MessageType: Enumeração que define os tipos de mensagens possíveis
    - SleepServiceDiscovery: Mensagem de descoberta.
    - SleepServiceMonitoring: Mensagem de monitoramento.
    - SleepServiceExit: Mensagem de saída.
    - SleepServiceCommand: Mensagem de comando.
    - Error: Mensagem de erro.
- CommandType: Enumeração que define os tipos de comandos possíveis:
    - Wakeup: Comando para acordar o participante.
    - Sleep: Comando para colocar o participante para dormir.

Atributos:

- type: Tipo da mensagem.
- seqn: Número de sequência da mensagem.
- length: Comprimento da mensagem.
- timestamp: Timestamp da mensagem.
- data: Pilha de dados da mensagem.

- src_ip: Endereço IP de origem da mensagem.

Métodos:

- Packet(int type, int seqn, int timestamp): Construtor do pacote na origem.
- Packet(char* buf): Construtor do pacote na chegada.
- push(std::string data): Adiciona dados ao pacote.
- pop(): Remove e retorna os dados do pacote.
- print(): Imprime o pacote.
- to_payload(): Converte o pacote em uma string de carga útil.
- get_type() const: Retorna o tipo da mensagem.

Funções Auxiliares:

- format_mac_address(const unsigned char* mac): Formata um endereço MAC.
- get_mac_address(): Obtém o endereço MAC do usuário atual.
- send_broadcast(Packet p, int sockfd, int port): Envia um pacote via broadcast.
- send_tcp(Packet p, int sockfd, int port, std::string ip = ""): Envia um pacote via TCP.
- rec_packet(int sockfd): Recebe um pacote via UDP.
- rec_packet_tcp(int sockfd): Recebe um pacote via TCP.

Arquivo host.cpp :

Este arquivo implementa as funções e métodos definidos no arquivo de cabeçalho host.h, fornecendo a lógica específica para as operações do participante no sistema.

Inicialização e Configuração:

- Host::init():
  - Inicializa os sockets de descoberta e monitoramento.
  - Cria e inicia as threads de descoberta, monitoramento, interface e entrada.
  - Utiliza mutexes para garantir operações seguras em threads concorrentes.
  - Configura a interface ncurses para exibir o status e a informação do participante.

Troca de Estado:

- Host::switch_state(HostState new_state):
  - Altera o estado do participante de forma segura, utilizando um mutex para evitar condições de corrida.
  - Atualiza o estado anterior e o novo estado.

Controlador de Saída do Participante:

- Host::exit_handler(int sn, siginfo_t* t, void* ctx):
  - Altera o estado do participante para Exit.

- Utiliza um mutex para garantir que a alteração do estado é segura.

Threads:

- Host::discovery(void *ctx):
  - Envia pacotes de descoberta em intervalos regulares.
  - Utiliza um loop para enviar pacotes até que o estado do participante mude para Exit.
- Host::monitoring(void *ctx):
  - Monitora o estado do participante e responde a comandos do manager.
  - Recebe pacotes via socket de monitoramento e executa ações baseadas no tipo de pacote recebido.
- Host::interface(void *ctx):
  - Atualiza a interface ncurses para exibir o estado e informações do participante.
  - Utiliza um loop para atualizar a interface até que o estado do participante mude para Exit.
- Host::input(void *ctx):
  - Lida com a entrada do usuário.
  - Permite ao usuário enviar comandos para mudar o estado do participante.

Arquivo manager.cpp :

Este arquivo implementa as funções e métodos definidos no arquivo de cabeçalho manager.h, fornecendo a lógica específica para as operações do manager no sistema.

Inicialização e Configuração:

- Manager::init():
  - Inicializa os sockets de descoberta e gerenciamento.
  - Cria e inicia as threads de descoberta, monitoramento, gerenciamento, comando, interface e entrada.
  - Utiliza mutexes para garantir operações seguras em threads concorrentes.
  - Configura a interface ncurses para exibir a lista de participantes conhecidos e suas informações.

Controlador de Saída do Manager:

- Manager::exit_handler(int sn, siginfo_t* t, void* ctx):
  - Cancela todas as threads e fecha os sockets ao receber um sinal de saída.
  - Utiliza mutexes para garantir que as operações de cancelamento e fechamento são seguras.

Gerenciamento de Participantes:

- Manager::add_host(KnownHost host): Adiciona um participante à lista de participantes conhecidos de forma segura, utilizando um mutex.
- Manager::remove_host(KnownHost host): Remove um participante da lista de participantes conhecidos de forma segura, utilizando um mutex.
- Manager::get_hosts(): Retorna uma cópia da lista de participantes conhecidos de forma segura, utilizando um mutex.
- Manager::has_host(std::string name): Verifica se um participante com um determinado nome está na lista de participantes conhecidos de forma segura, utilizando um mutex.
- Manager::check_input(std::string input): Verifica e retorna um comando e o nome do participante correspondente a partir de uma string de entrada.

Threads:

- Manager::discovery(void *ctx):
  - Descobre participantes na rede, recebendo pacotes de descoberta via socket de descoberta.
  - Adiciona participantes descobertos à lista de participantes conhecidos.
- Manager::monitoring(void *ctx):
  - Monitora os participantes conhecidos, recebendo pacotes de monitoramento via socket de gerenciamento.
  - Atualiza o estado dos participantes conhecidos com base nos pacotes recebidos.
- Manager::management(void *ctx):
  - Gerencia as conexões com os participantes conhecidos, enviando pacotes de comando para acordar ou colocar participantes para dormir.
- Manager::command(void *ctx):
  - Processa comandos, enviando pacotes de comando para os participantes conhecidos.
- Manager::interface(void *ctx):
  - Atualiza a interface ncurses para exibir a lista de participantes conhecidos e suas informações.
  - Utiliza um loop para atualizar a interface até que o estado do manager mude para Exit.
- Manager::input(void *ctx):
  - Lida com a entrada do usuário.
  - Permite ao usuário enviar comandos para gerenciar os participantes conhecidos.

Arquivo message.cpp :

Este arquivo implementa as funções definidas no arquivo de cabeçalho message.h, fornecendo a lógica para manipulação de pacotes de rede.

Construtores:

- Packet::Packet(int type, int seqn, int timestamp): Construtor para criar um pacote na origem, inicializando os atributos type, seqn, timestamp e length.
- Packet::Packet(char* buf): Construtor para criar um pacote na chegada, inicializando os atributos a partir de um buffer recebido.

Manipulação de Dados:

- Packet::push(std::string data): Adiciona dados ao pacote, atualizando o comprimento do pacote.
- Packet::pop(): Remove e retorna os dados do pacote, atualizando o comprimento do pacote.
- Packet::print(): Imprime os detalhes do pacote para fins de depuração.
- Packet::to_payload(): Converte o pacote em uma string de carga útil para envio.
- Packet::get_type() const: Retorna o tipo da mensagem do pacote.

Funções Auxiliares

- format_mac_address(const unsigned char* mac): Formata um endereço MAC em uma string legível.
- get_mac_address(): Obtém o endereço MAC do usuário atual.
- send_broadcast(Packet p, int sockfd, int port): Envia um pacote via broadcast utilizando um socket.
- send_tcp(Packet p, int sockfd, int port, std::string ip = ""): Envia um pacote via TCP utilizando um socket.
- rec_packet(int sockfd): Recebe um pacote via UDP utilizando um socket.
- rec_packet_tcp(int sockfd): Recebe um pacote via TCP utilizando um socket.

Explicar o uso das diferentes primitivas de comunicação

Neste projeto, foram utilizadas as primitivas de comunicação UDP e TCP para implementar os subserviços de descoberta e monitoramento. Aqui está uma explicação de como essas primitivas foram utilizadas de forma geral:

Sockets UDP (User Datagram Protocol)

- Subserviço de Descoberta:

  Envio de Broadcasts: Utilizado para enviar mensagens de broadcast na rede, permitindo que todos os hosts na mesma sub-rede recebam a mensagem.

  Recepção de Broadcasts: Permite que o manager receba essas mensagens e identifique os hosts ativos na rede.

Exemplo de Uso no Host:

O socket UDP é criado e configurado para permitir o envio de broadcasts. A configuração adequada das opções do socket garante que ele possa enviar mensagens de broadcast na rede.

Exemplo de Uso no Manager:

O socket UDP é criado e configurado para reutilizar o endereço local. Isso permite que o manager receba as mensagens de broadcast enviadas pelos hosts.

Sockets TCP (Transmission Control Protocol)

- Subserviço de Monitoramento:

  Conexões: Utilizado para estabelecer conexões confiáveis entre o manager e os hosts. Isso permite que o manager monitore o estado dos hosts e envie comandos.

  Envio e Recepção de Broadcast: Permite o envio de pacotes de monitoramento e a recepção de respostas dos hosts, contendo o estado de cada host.

Exemplo de Uso no Host:

O socket TCP é criado e configurado para permitir conexões. Isso garante que o host possa aceitar conexões do manager para receber comandos e enviar seu estado atual.

Exemplo de Uso no Manager:

O socket TCP é criado e utilizado para conectar-se aos hosts. Isso permite que o manager envie pacotes de monitoramento e receba respostas, atualizando o estado dos hosts conforme necessário.

Dificuldades encontradas

Durante a execução deste trabalho, encontramos algumas dificuldades que impactaram o desenvolvimento e implementação dos subserviços propostos. Abaixo, listamos as principais dificuldades enfrentadas pela equipe:

1. Sincronização de Threads

Uma das maiores dificuldades foi garantir a sincronização adequada entre as threads. Implementar primitivas de exclusão mútua corretamente foi desafiador, pois qualquer erro na sincronização poderia levar a condições de corrida ou deadlocks. Foi necessário um esforço significativo para identificar todas as seções críticas e garantir que mutexes fossem utilizados corretamente.

2. Gerenciamento de Sockets

Gerenciar múltiplos sockets simultaneamente foi complexo, especialmente no que diz respeito à criação, reutilização e encerramento dos mesmos. Lidamos com problemas como a necessidade de manipular corretamente as conexões TCP e UDP. A implementação de loops de descoberta e monitoramento de forma eficiente e sem desperdício de recursos também exigiu bastante atenção.

3. Testes e Depuração

Testar e depurar um sistema com múltiplas threads e processos concorrentes foi extremamente desafiador. Condições de corrida e bugs intermitentes foram difíceis de reproduzir e diagnosticar.

4. Manutenção da Lista de Participantes

Manter a lista de participantes de forma consistente e atualizada foi outro desafio significativo. A lista precisava ser protegida contra acesso simultâneo e, ao mesmo tempo, ser suficientemente eficiente para não causar um gargalo no sistema. Implementamos vários mecanismos de mutexes e técnicas de cópia de listas para garantir a integridade dos dados.

Conclusão

Apesar das várias dificuldades encontradas, conseguimos superá-las e implementar um sistema funcional para gerenciamento de participantes utilizando threads, sincronização e comunicação em rede. Este trabalho proporcionou um aprendizado significativo sobre os desafios e técnicas envolvidas no desenvolvimento de sistemas concorrentes e distribuídos.