Trabalho Prático 1 - Documentação
Redes de Computadores

Mateus Brandão Damasceno Góes | 2020054706

1. Introdução

Este trabalho tem como objetivo implementar um sistema de controle de acesso aos prédios universitários utilizando catracas inteligentes simuladas, estabelecendo a comunicação entre interfaces de controle (ICs) e servidores através do uso de sockets. Para alcançar esse objetivo, foram desenvolvidos dois tipos de executáveis: um servidor, que pode atuar como servidor de usuários (SU) ou servidor de localização (SL), e um cliente que representa as interfaces de controle. O sistema implementa uma arquitetura onde os servidores se comunicam entre si através de uma conexão peer-to-peer, enquanto mantêm conexões simultâneas com múltiplos clientes.

Os servidores são responsáveis por diferentes aspectos do sistema: o servidor de usuários (SU) gerencia o cadastro e autenticação das pessoas, enquanto o servidor de localização (SL) mantém o registro da posição atual de cada indivíduo dentro do campus. A comunicação entre estes componentes acontece através de um protocolo próprio implementado sobre TCP, garantindo a confiabilidade na troca de mensagens. As interfaces de controle, por sua vez, atuam como pontos de acesso onde os funcionários podem realizar operações como cadastro de usuários, consulta de localização e controle de entrada e saída dos prédios.

O desenvolvimento foi realizado em linguagem C utilizando a biblioteca padrão POSIX para comunicação via sockets, com suporte tanto para IPv4 quanto IPv6. A implementação foca em criar uma base robusta para o sistema de controle de acesso. As seções seguintes detalharão a arquitetura do sistema, o protocolo de comunicação utilizado, e as decisões de implementação adotadas para atender aos requisitos especificados.

2. Mensagens

O protocolo de comunicação implementado no sistema utiliza mensagens estruturadas com tamanho máximo de 500 bytes, transmitidas através de conexões TCP. A estrutura base das mensagens foi implementada através de uma struct *Message*, que contém três campos principais: type (identificador do tipo da mensagem), size (tamanho do payload), e payload (conteúdo da mensagem). Esta estrutura permite uma comunicação uniforme entre todos os componentes do sistema, facilitando o processamento e a interpretação das mensagens.

Para garantir a interpretação das mensagens em todo o código, foi implementada uma função *setMessage* que padroniza a criação das mensagens, preenchendo os campos necessários e garantindo que o payload não ultrapasse o limite estabelecido. O processamento das mensagens é realizado através da função *processServerMessage*, que analisa o tipo da mensagem recebida e executa as ações correspondentes. Esta função atua como um distribuidor central, direcionando as mensagens para as funções específicas de processamento de acordo com seu tipo.

Um aspecto importante da implementação foi o tratamento das mensagens entre os servidores peer-to-peer. Quando uma operação requer a comunicação entre o servidor de usuários e o servidor de localização, como no caso da autenticação de usuários para consulta de localização, as mensagens são encaminhadas através da conexão peer-to-peer estabelecida. Para isso, foi criada uma thread dedicada (*handlePeerConnection*) que gerencia esta comunicação, garantindo que as mensagens sejam processadas de forma assíncrona sem bloquear as demais operações.

3. Arquitetura

O sistema é composto por três componentes principais: as interfaces de controle (ICs), o servidor de usuários (SU) e o servidor de localização (SL). Ambos os servidores são iniciados através do mesmo programa do server.

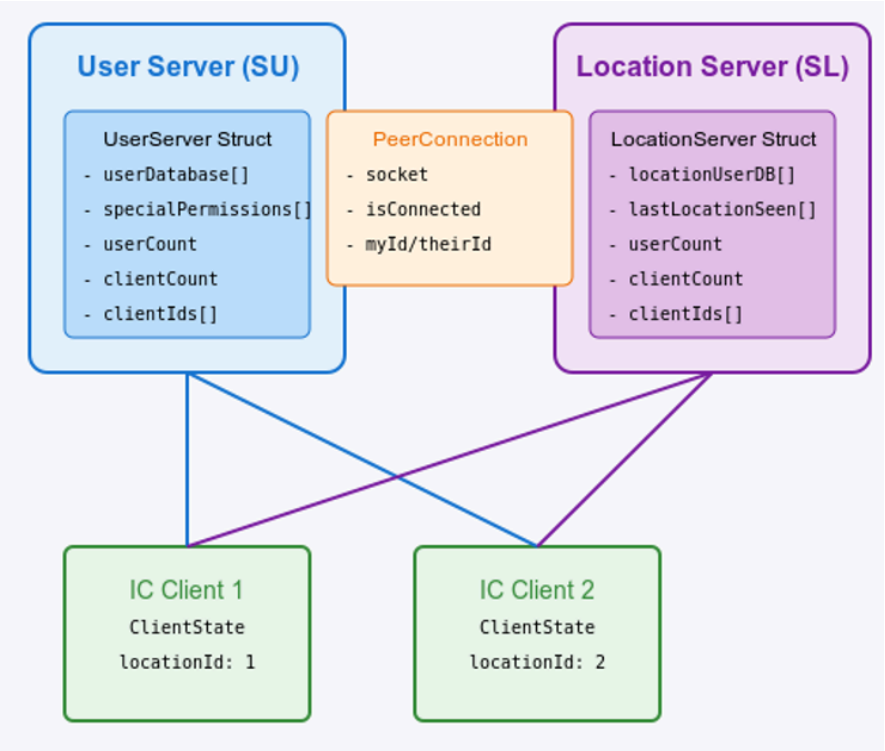
Para gerenciar o estado dos servidores, implementei duas estruturas principais: *UserServer* e *LocationServer*. A estrutura *UserServer* mantém arrays para armazenar os IDs dos usuários cadastrados e suas permissões especiais, além de controlar a quantidade de clientes conectados. De forma similar, a estrutura *LocationServer* armazena o registro das últimas localizações dos usuários e mantém seu próprio controle de clientes. Ambas as estruturas foram projetadas para suportar até 30 usuários cadastrados e 10 conexões simultâneas de clientes.

A comunicação peer-to-peer entre os servidores foi implementada através da estrutura *PeerConnection*, que mantém informações sobre o estado da conexão, identificadores únicos para cada servidor e ponteiros para as estruturas de dados relevantes, que são preenchidas de acordo com o tipo de servidor que *PeerConnection* é criado. Essa *struct* é usada como controle para a identificação se a conexão está ativa, o armazenamento dos Ids de cada servidor, a porta e o socket conectado atualmente em cada server.

O gerenciamento de múltiplas conexões foi implementado utilizando a função select(), permitindo que os servidores monitorem simultaneamente várias conexões de clientes e a conexão peer-to-peer. Por motivos de problemas concorrentes entre o gerenciamento de vários selects achei válido utilizá-los para a seleção dos sockets dos clientes e a manutenção das diversas conexões, enquanto que também utilizei de threads para a comunicação peer-to-peer (*handlePeerConnection*), para processar os comandos no terminal do servidor *(handleServerStdin)* e para os clientes conectados no servidor *(handleClientMessages)*. Essa abordagem do uso tanto de select e threads foi de bastante ajuda, já que os selects estavam ficando bem confusos e pela minha familiaridade com o uso de threads.

Além disso, foi implementado um sistema de estados para os clientes através da estrutura *ClientState*, que mantém o controle dos identificadores atribuídos pelos servidores e os status da inicialização. Esta abordagem permite que os clientes mantenham conexões independentes com os dois servidores, enquanto garantem a sincronização de informações.

Abaixo está uma representação de como a arquitetura dos servidores/clientes foi organizada contendo as principais estruturas utilizadas:



4. Servidor

A implementação do servidor foi desenvolvida de forma a atender dois papeis distintos: servidor de usuários (SU) e servidor de localização (SL). O código foi estruturado de maneira que um mesmo executável possa exercer qualquer uma dessas funções, determinada através das portas fornecidas como parâmetro na execução.

O servidor inicia sua execução estabelecendo dois sockets principais: um para a comunicação peer-to-peer e outro para atender as conexões dos clientes. A inicialização destes sockets é realizada através das funções *setupPeerServerSocket* e *setupClientServerSocket*, que configuram os parâmetros necessários para suportar tanto IPv4 quanto IPv6, essa configuração foi feita iniciando o socket como IPv6 e desabilitando a flag IPV6_V6ONLY, dessa forma permitindo também conexões IPv4.

O servidor então tenta estabelecer uma conexão peer-to-peer inicial através da função *establishPeerConnection*. Se esta tentativa falhar, indicando que não há outro servidor disponível, o servidor configura um socket de escuta peer-to-peer através de *setupPeerServerSocket* e imprime a mensagem "No peer found, starting to listen...". Isso permite que o primeiro servidor fique aguardando a conexão do segundo.

Após a configuração inicial, o servidor cria duas threads: uma para monitorar comandos do terminal através da função *handleServerStdin*, que processa principalmente o comando "kill" para encerramento do servidor, e outra para gerenciar a comunicação peer-to-peer através de *handlePeerConnection*, que lida com todas as mensagens trocadas entre os servidores.

O loop principal do servidor utiliza select() para monitorar simultaneamente o socket de clientes e, quando aplicável, o socket peer-to-peer. Quando uma nova conexão de cliente é detectada, o servidor realiza as seguintes etapas:

1. Aceita a conexão e verifica se o limite de clientes (10) foi atingido
2. Aguarda e processa a mensagem inicial REQ_CONN do cliente

3. Cria uma estrutura *ClientThreadParams* contendo os parâmetros necessários para o processamento das mensagens
4. Inicia uma nova thread executando *handleClientMessages* para processar todas as futuras mensagens deste cliente

Esta arquitetura permite que o servidor realize atividades distintas usando diferentes métodos de controle. Além disso, a função *handleClientMessages* é executada em uma thread separada para cada cliente, processando suas requisições de forma independente.

O processamento das mensagens é centralizado na função *processServerMessage*, que atua como um dispatcher, direcionando as mensagens recebidas para as funções específicas de tratamento. Esta função implementa toda a lógica de negócio do servidor, incluindo o cadastro de usuários, controle de acesso e consultas de localização. Para garantir a consistência dos dados, foram usadas as estruturas de controle como *UserServer* e *LocationServer*, que mantêm o estado atual do sistema e são atualizadas de forma thread-safe.

5. Cliente

A implementação do cliente representa as Interfaces de Controle (ICs) do sistema. O cliente foi desenvolvido para manter conexões simultâneas com ambos os servidores (SU e SL).

O cliente inicia sua execução validando os parâmetros de entrada, que incluem o endereço do servidor, as portas dos servidores SU e SL, e o código de localização. Uma verificação implementada é a validação do código de localização através da função *validateLocationId*, que garante que apenas valores entre 1 e 10 sejam aceitos. Esta validação é realizada antes de qualquer tentativa de conexão com os servidores, evitando o estabelecimento desnecessário de conexões quando os parâmetros são inválidos.

Após a validação dos parâmetros, o cliente estabelece as conexões com ambos os servidores utilizando a função *initializeClientSockets*. Uma vez estabelecidas as conexões, o cliente envia uma mensagem inicial *REQ_CONN* para cada servidor, contendo seu código de localização. Os servidores respondem com identificadores únicos que são armazenados na estrutura *ClientState*, responsável por manter o estado interno do cliente.

O loop principal do cliente utiliza select() para monitorar simultaneamente três fontes de eventos: a entrada padrão (comandos do usuário) e os dois sockets dos servidores. Quando um comando é recebido pela entrada padrão, a função *parseUserCommand* realiza o parsing e validação do comando, preparando a mensagem apropriada para ser enviada ao servidor correto. Os comandos suportados incluem operações como cadastro de usuários (add), consulta de localização (find), registro de entrada e saída (in/out) e consulta de pessoas em uma localização específica (inspect).

O processamento das respostas dos servidores é realizado pela função handleReceivedData, que interpreta as mensagens recebidas e apresenta as informações adequadas ao usuário através da saída padrão. Esta função implementa tratamentos específicos para cada tipo de mensagem, incluindo confirmações de operações bem-sucedidas e tratamento de erros como limite de clientes excedido, usuário não encontrado e permissão negada.

Para o encerramento de conexão através do comando kill, o cliente garante que ambos os servidores sejam notificados apropriadamente. Além disso, o cliente possui tratamento de erros para situações como limite de clientes excedido, usuário não encontrado e permissão negada, apresentando as mensagens ao usuário em cada caso.

5. Discussão/Conclusão

No geral, esse foi um trabalho bem desafiador que levou meses para ser desenvolvido de forma completa, mas acredito que me ajudaram a entender mais os conceitos de programação em rede. Uma das partes mais desafiadoras foi a implementação do server peer-to-peer e das múltiplas conexões de clientes, uma vez que estava com muitos problemas no uso do select para essa etapa.

Dessa forma, o uso de threads junto com o select me ajudou bastante. No início foi difícil porque depois que tive os problemas iniciais usando o select e os FDs tive que recomeçar a implementação das múltiplas conexões usando threads e no final acredito que a utilização de threads separadas para a comunicação peer-to-peer e para o processamento de comandos do terminal permitiu uma melhor organização do código e uma separação mais clara das responsabilidades.

Além disso, para a implementação dos comandos de cliente e servidor a etapa do fluxo das mensagens de controle que estava descrito na especificação do trabalho foi bastante útil. Ter o fluxo descrito para cada tipo de mensagem ajudou na depuração e na implementação inicial de cada etapa, separando parte por parte e ajudando a quebrar o trabalho em problemas menores.

Acredito que todas as funcionalidades foram implementadas de acordo com o fluxo demonstrado e os testes foram feitos baseados nos exemplos de execução disponibilizados. Um ponto que poderia ser levado em consideração seria a disponibilização dos testes automatizados para os alunos ou mais exemplos de edge cases possíveis, uma vez que cada mensagem trocada com o servidor tem inúmeras possibilidades de divergência, mas no geral, os exemplos ajudaram a pensar em testes possíveis.

Acredito também que não houveram decisões de implementações diferentes das documentadas e procurei adicionar mais comentários no código caso seja encontrada alguma divergência para que meu raciocínio seja explicado pelo código caso seja necessário.

Por fim, o trabalho foi um bom desafio e ampliou meus conhecimentos em desenvolvimento de programação com sockets e sistemas distribuídos.