# 1. Introdução

Este relatório apresenta a implementação de um sistema de comunicação cliente-servidor desenvolvido em C, utilizando sockets. O sistema foi projetado para gerenciar o controle de acesso por meio de catracas inteligentes e rastrear a localização de indivíduos dentro da universidade. Ele permite a comunicação eficiente entre múltiplos clientes e dois servidores, onde cada cliente representa um prédio do campus universitário e cada servidor tem uma responsabilidade. O primeiro é o servidor de usuário responsável por armazenar os dados dos usuários, como identificadores e registros de permissão. Já o segundo é o servidor de localização, encarregado de registrar as informações de localização dos usuários, incluindo seus identificadores e a última posição registrada.

As principais funcionalidades do sistema incluem: registro de usuários, gerenciamento de permissões de acesso, registro de localizações, comunicação peer-to-peer entre os servidores e comunicação com múltiplos clientes. O sistema conta com o apoio do protocolo TCP para realizar o transporte das mensagens. Este relatório detalha o desenvolvimento, as características técnicas e os resultados obtidos na implementação desse sistema.

# 2. Mensagens

Foram especificadas quatro categorias de mensagens: 6 mensagens de controle, 11 mensagens de dados, 7 tipos de mensagens de erro e 3 tipos de mensagens de sucesso. A seguir um exemplo de como o pacote de mensagem foi estruturado, de cada tipo de mensagem descrita anteriormente.

| **Mensagens de Controle e Dados** | |
|---|---|
| **Tipo e (Direção)** | **Pacote** |
| RES_CONNPEER (Sj->Si ) | “18 PidSi” |

| **Mensagens de Erro** | |
|---|---|
| **Tipo e (Direção)** | **Pacote** |
| Error 01 | “255 Peer limit exceeded” |

| **Mensagens de Confirmação** | |
|---|---|
| **Tipo e (Direção)** | **Pacote** |
| OK 01 | “0 Successful disconnect” |

As mensagens são enviadas por meio de um buffer de caracteres com capacidade máxima de 1024 posições. Cada elemento do buffer é do tipo char, que ocupa 1 byte (conforme a definição na maioria dos sistemas baseados em C).

$$Tamanho\ total\ do\ buffer\ =\ 1024\ x\ 1\ byte\ =\ 1024\ bytes.$$

Conforme descrito no enunciado, cada mensagem pode ter no máximo 500 bytes. Assim, o buffer de envio de dados é adequado para suportar todas as mensagens consideradas neste trabalho.

# 3. Arquitetura

O sistema é composto por 3 arquivos principais de código:

- **client.c:** Implementação do cliente.
- **server.c:** Implementação dos servidores SU e SL.
- **common.c:** Implementação de funções que possam ser reaproveitadas pelos dois arquivos.
- **commom.h:** Definição das funções presentes no arquivo *common.c* e dos *defines* que especificam os códigos das respectivas mensagens de controle, dados, erro e confirmação, além do tamanho padrão do buffer.

O arquivo que contém a implementação do servidores contém as seguintes estruturas de dados:

**1. struct user**

**Conceito:** Cada usuário representa uma pessoa que se desloca pelos prédios da universidade. O servidor de usuários suporta até 30 pessoas cadastradas, e os dados relacionados a esses usuários são armazenados em um vetor com capacidade para 30 posições. Como o sistema realiza apenas operações de cadastro e consulta, sem remoção de usuários, o uso desse vetor é adequado. Além disso, há um controle implementado para garantir que não sejam adicionados mais de 30 usuários ao servidor, assegurando a integridade do armazenamento.

**Função:** Armazena informações sobre os usuários, incluindo *userId*, *hasPermission* e *locId*.

**2. struct client**

**Conceito:** Cada cliente representa um computador instalado em um prédio específico da universidade. Ao ser inicializado, o cliente comunica o identificador de sua localização, que é um número entre 1 e 10. O sistema suporta até 10 clientes conectados simultaneamente. No entanto, os clientes podem encerrar suas conexões, deixando seus registros obsoletos. Para gerenciar essa dinâmica, os dados são armazenados em um vetor com capacidade para até 1024 posições, permitindo registrar até 1024 interações de conexão e desconexão em um único servidor.

**Função:** Armazena informações sobre os clientes, incluindo *clientId*, *locId* e *socket,* este último é o número do socket.

# 4. Servidor

O servidor implementado neste trabalho suporta dois tipos de conexões: com clientes e com outro servidor. Para isso, durante a inicialização, o programa recebe os seguintes argumentos:

*./server portConexãoPeer portConexãoCliente*

- **portConexãoPeer:** Porta destinada à comunicação com outro servidor.

- **portConexãoCliente:** Porta destinada à comunicação com os clientes.

Com base nas portas especificadas, o servidor cria dois sockets: um para conexões com clientes e outro para a conexão peer (entre servidores). O socket é responsável por informar ao sistema operacional que o programa realizará comunicações em rede em uma porta específica, utilizando um determinado protocolo.

Para garantir a compatibilidade com diferentes protocolos, o socket foi configurado para a família de endereços IPv6, permitindo também conexões IPv4. Isso foi alcançado desabilitando a opção **IPV6_V6ONLY** no socket com o uso da função setsockopt.

Além disso, o socket foi configurado para reutilizar endereços ou portas, o que possibilita o reinício rápido do servidor sem a necessidade de esperar pela liberação completa dos recursos.

Por fim, o identificador do socket criado e configurado é retornado, permitindo que ele seja usado em operações futuras, como associar o socket a uma porta específica **(bind)**, começar a escutar conexões **(listen)** e aceitar conexões de entrada **(accept)**. Essa mesma operação foi feita para a criação de ambos os sockets, um para conexões de clientes e outro para conexão peer. Esse identificador é salvo em duas variáveis, *socket_response* para conexões com clientes e *socket_response_peer* para conexão peer.

Após isso, foi feita uma abertura passiva, constituída pelas operações bind e listen, para aceitar novos clientes. Em seguida, foi feita uma abertura passiva para receber conexões com servidores. Para a comunicação peer, existe uma regra de operação: inicialmente, o servidor tenta realizar a conexão passiva na porta especificada para a comunicação peer. Se já houver uma conexão passiva ativa nessa porta, a operação **bind** retornará um erro. Nesse caso, o servidor realizará uma tentativa de conexão ativa na mesma porta. Caso a conexão for aceita, esse servidor já solicita o pedido de conexão com a mensagem REQ_CONNPEER e trata a resposta dessa mensagem RES_CONNPEER.

Dessa forma, o primeiro servidor a ser iniciado realiza uma abertura passiva, ficando aguardando novas conexões peer. Já o segundo servidor, ao ser iniciado, fará uma conexão ativa na porta especificada, conectando-se ao primeiro servidor, que estará escutando novas conexões.

As estruturas de dados responsáveis por armazenar as informações dos usuários e clientes foram alocadas. O parâmetro locId dos usuários foi inicializado com o valor -1, pois, ao ser criado, um usuário ainda não está associado a nenhum prédio.

Para gerenciar múltiplas conexões simultâneas, foi utilizada a função **select()**. O **select()** monitora a atividade dos sockets: o socket para comunicação com clientes, o socket para comunicação com o peer e o STDIN para leitura da entrada padrão pelos servidores.

Quando um socket apresenta atividade, a função verifica se o socket em questão é o utilizado para comunicação com os clientes. Se for, isso indica que um cliente está tentando se conectar. Nesse caso, a conexão é aceita e o número de conexões ativas é verificado. Se o número de conexões exceder 10, a conexão é encerrada. Caso contrário, o socket retornado pela operação **accept()** é adicionado ao conjunto de sockets a ser monitorado pelo **select()**.

Se o socket com atividade for o utilizado para comunicação com o peer, verifica-se se a conexão passiva foi aberta no servidor. Se a conexão passiva estiver aberta, isso indica que um novo servidor está tentando se conectar ao servidor que está aguardando novas conexões (ou seja, o servidor ativo fez uma solicitação ao servidor passivo). Caso já exista um servidor conectado, a nova conexão é recusada. Se não houver outra conexão, a nova conexão é aceita e o socket retornado pela operação **accept()** é adicionado ao

conjunto de sockets a ser monitorado pelo **select()**. Esse socket é salvo na variável *connection_socket_peer* para uso posterior.

Se a atividade for no **STDIN**, a entrada padrão é lida e a operação correspondente ao comando digitado é executada. Se a atividade não estiver relacionada a nenhum dos sockets mencionados anteriormente, isso indica que o evento ocorreu em um dos sockets adicionados pelo accept(). Isso pode corresponder a um socket de cliente (seja no servidor SU ou SL), ao socket do servidor que se conectou ativamente ao outro na comunicação peer ou ao socket aberto para comunicação peer.

No último caso, temos o seguinte detalhamento das três situações listadas:

- **Situação 1:** A atividade ocorreu no socket gerado pelo accept() ao aceitar uma conexão com outro peer. Nesse contexto, é necessário verificar se o servidor atual é aquele que abriu a conexão de forma passiva. Em outras palavras, isso significa que o servidor Sj abriu a conexão passivamente, enquanto o servidor Si estabeleceu a conexão de forma ativa. O servidor Sj aceitou a conexão e armazenou o socket gerado pelo accept() no conjunto de descritores. Se esse socket apresentar atividade e o servidor em execução for Sj, isso indica que o servidor Si está enviando uma mensagem para o servidor Sj. Nesse caso, é necessário tratar as mensagens geradas por esse fluxo, que podem incluir: REQ_CONPEER, RES_CONPEER, REQ_DISCPEER e REQ_USRAUTH.
- **Situação 2:** A atividade foi detectada no socket associado à conexão peer, armazenado na variável *socket_response_peer*, e o servidor em execução é aquele que estabeleceu a conexão de forma ativa. Em outras palavras, o servidor Sj, que abriu a conexão passivamente, está enviando uma mensagem para o servidor Si, que realizou a conexão de forma ativa. Como o servidor em execução é o Si, ele deve processar mensagens do tipo REQ_LOCREG.
- **Situação 3:** Se não for a Situação 1 ou a Situação 2, isso indica que algum cliente está enviando uma mensagem, seja para o servidor Sj ou para o servidor Si. Nesse caso, as seguintes mensagens devem ser processadas: REQ_CONN, REQ_DISC, REQ_USRADD, REQ_USRLOC, REQ_USRACCESS e REQ_LOCLIST.

# 5. Cliente

O cliente implementado neste trabalho suporta conexão com dois servidores. Para isso, durante a inicialização, o programa recebe os seguintes argumentos:

./client endereço portConexãoServidor1 portConexãoServidor2 locId

- **endereço:** endereço IPV4 ou IPV6 do cliente.
- **portConexãoServidor1:** Porta destinada à comunicação com o servidor 1.
- **portConexãoServidor2:** Porta destinada à comunicação com o servidor 2.
- **locId:** Identificador da localização do prédio que está contido o referido cliente.

Primeiramente, verifica-se se a localização fornecida está entre 1 e 10. Caso contrário, será exibida a mensagem *Invalid argument*, e o programa será encerrado sem execução. Se a localização for válida, dois sockets são abertos: um para comunicação com o servidor de usuário **(SU)** e outro com o servidor de localização **(SL)**. Em seguida, estabelece-se a comunicação ativa com ambos os servidores.

Se o cliente receber a resposta de conexão de ambos os servidores, ele imediatamente envia a mensagem REQ_CONN para os dois e aguarda suas respectivas respostas. Caso o cliente não consiga estabelecer conexão com algum dos servidores, uma mensagem de erro será exibida, e o programa do cliente será encerrado. Se o cliente estabelecer corretamente a conexão com ambos os servidores, ele receberá seu

identificador na rede e estará habilitado a receber entradas pelo teclado, permitindo a execução das operações necessárias.

O cliente está configurado para receber e executar os seguintes comandos: kill, add UID IS_SPECIAL, find UID, in UID, out UID e inspect UID LOC. Caso seja enviado um comando diferente dos listados ou se os argumentos de qualquer comando estiverem incorretos ou incompletos, será exibida a mensagem *Invalid argument*, e o comando não será processado. O usuário poderá, então, inserir novos comandos corretamente posteriormente.

# 6. Discussão

### *6.1. Descrição dos Resultados:*

Foram implementadas **todas as funcionalidades pedidas**, tanto para IPV4 quanto para IPV6. Os testes foram executados localmente, via linha de comando e analisadas as respectivas saídas para os fluxos de mensagens dispostos. Foi testado o fluxo de mensagem especificado no arquivo **Exemplos de Execução Trabalho Prático**, disponibilizado pelo monitor.

### *6.2. Desafios e Soluções*

O primeiro desafio foi, sem dúvida, compreender o funcionamento da comunicação entre o servidor e o cliente. Essa foi a etapa inicial do trabalho, que envolveu a implementação da troca de mensagens entre um servidor e um cliente, ou seja, a manipulação de sockets, a abertura de comunicação passiva e ativa, e o uso das funções **send** e **recv**. Para que isso fosse possível, foi necessário ler os argumentos da entrada padrão, interpretá-los e manipulá-los. Durante esse processo, encontrei vários obstáculos relacionados à linguagem C, especialmente no que diz respeito à manipulação de strings, principalmente no envio e leitura dos pacotes trocados entre o cliente e o servidor. O desafio consistia em inserir dados dinâmicos nas mensagens a serem enviadas e recuperar dados das mensagens recebidas. Ao longo desse processo, aprendi a utilizar funções de manipulação de strings que eu desconhecia anteriormente. Para facilitar a formatação dos pacotes de dados, foram utilizados defines no arquivo *common.h*, que associam cada mensagem ao seu respectivo código.

Após superar esse desafio, passei para a próxima etapa: fazer dois servidores se comunicarem. Nesse ponto, a leitura do fórum foi fundamental, pois o monitor forneceu uma dica crucial: o primeiro servidor a ser iniciado será configurado para uma conexão passiva, enquanto o segundo servidor, ao tentar abrir passivamente, encontrará um erro no bind, o que indicará que ele deve ser iniciado de forma ativa.

Depois de superar essa etapa, o maior desafio e o mais cansativo foi entender o uso da função select para monitorar múltiplas conexões, tanto de clientes quanto de servidores. Embora o uso do select não seja estritamente necessário, foi importante entender o conceito de descritores e como os sockets podem ser monitorados por meio desses descritores. Uma vez compreendido esse conceito, ficou mais fácil implementar a lógica de monitoramento dos descritores, determinar qual socket estava ativo e definir a ação a ser tomada conforme a atividade de cada socket.

Ademais, enfrentei dificuldades para modularizar o código, pois muitos dados precisavam ser passados como parâmetros, alguns por referência, o que tornou o processo de organização e estruturação do código mais desafiador.

Por fim, um dos principais desafios enfrentados foi implementar a lógica de fechamento de conexão entre os servidores peer. Esse processo é complexo porque a desconexão pode ser iniciada em ambas as direções, além de exigir que as conexões dos clientes sejam encerradas quando um servidor decide fechar sua conexão. Para resolver esse problema, foi adotada a seguinte abordagem:

1. **Para os dois servidores:**

Quando o servidor sj envia um pedido de desconexão ao servidor si, ocorre a troca necessária de mensagens para a finalização da conexão. Após isso, o servidor sj é encerrado. Em seguida, quando o servidor si conclui o processamento das mensagens de desconexão, ele desconecta todos os clientes conectados, iterando por seus sockets. Durante esse processo, o número de conexões ativas é decrementado. Por exemplo, se o servidor si tinha 4 conexões ativas, ao final do procedimento, ele ficará com 0, pois terá encerrado todas as conexões com os clientes.

2. **Caso o servidor conectado passivamente solicite a desconexão:**

Nesse cenário, o servidor que estava ativo na conexão deve iniciar o processo de escuta para novas conexões. Assim, quando o servidor que estava ativo conclui o tratamento das mensagens de desconexão, ele tenta se conectar passivamente e começa a escutar novas conexões.

### *6.3. Limitações Identificadas*

A principal limitação identificada durante o desenvolvimento foi o tamanho do código e a falta de modularização adequada. Como o trabalho permitia a implementação de apenas três tipos de arquivos, o código final acabou se tornando extenso e difícil de manter. Como resultado, o código pode não seguir as melhores práticas de programação tanto quanto deveria. Uma das questões mais evidentes é o uso excessivo de estruturas condicionais **if** em várias partes do código, o que compromete a legibilidade e dificulta a manutenção. Embora o código tenha sido refatorado várias vezes ao longo do desenvolvimento, acredito que ainda há espaço para melhorias, especialmente no que diz respeito à clareza e modularização, tornando-o mais limpo e organizado.

Além disso, o sistema não é tão otimizado em relação à alocação de memória para os recursos utilizados durante a execução, o que é uma preocupação importante, considerando que C é uma linguagem de baixo nível que oferece grande flexibilidade para otimizar o uso da memória. A alocação de memória poderia ser mais eficiente, especialmente em casos de grandes volumes de dados ou quando o sistema precisar de escalabilidade.

Outro ponto que poderia ser melhorado é a consistência na adoção de uma convenção de nomenclatura mais rigorosa. Isso ajudaria a tornar o código mais intuitivo, facilitando a leitura e a colaboração entre desenvolvedores. Em resumo, embora o sistema tenha alcançado os objetivos propostos, existem várias áreas em que melhorias poderiam ser feitas para aprimorar a qualidade do código e a eficiência do sistema como um todo.

Por fim, devido à complexidade do sistema e ao grande esforço envolvido em sua implementação, optou-se por não realizar uma refatoração completa do código, a fim de evitar a introdução de possíveis bugs e garantir a estabilidade do sistema.

# 7. Conclusão

Este relatório descreve a implementação de um sistema de comunicação cliente-servidor em C. O desenvolvimento deste sistema apresentou desafios significativos, mas foi uma oportunidade valiosa para compreender, na prática, o funcionamento de um protocolo em ação. Embora o protocolo TCP seja amplamente utilizado no dia a dia, muitas vezes não temos uma visão clara de como ele opera internamente. Este trabalho permitiu explorar o funcionamento específico do TCP e os fundamentos de um protocolo de comunicação entre máquinas distintas.

A troca de informações segue regras bem definidas, evidenciando a importância de especificar com precisão quais mensagens serão transmitidas, como serão estruturadas e transportadas, e como tratar cada caso de forma adequada. Essa experiência proporcionou um entendimento mais profundo sobre os mecanismos subjacentes à comunicação em rede. Este trabalho permitiu exercitar os três principais pilares de uma troca de mensagens: **Tipo** (definindo se é uma solicitação ou resposta), **Sintaxe** (estrutura e campos que compõem a mensagem) e **Semântica** (o significado do conteúdo dos campos e como interpretá-los adequadamente).

Em resumo, com esse trabalho foi possível exercitar vários conceitos e aplicações aprendidos na disciplina de Redes de Computadores por completo.