

# Tutorial

Ricardo de la Rocha Ladeira

## Introdução

Na [aula passada](#) foram trabalhados aspectos de [sincronização de processos e threads](#), envolvendo os conceitos de condição de corrida, seção crítica e estratégias de sincronização em hardware e software.

Aproveitando a base obtida pelas aulas passadas, a proposta deste material é apresentar um passo a passo para a criação de uma aplicação *multithread*. A aplicação escolhida é um programa de bate-papo.

Portanto, o objetivo desta aula é colocar em prática os conhecimentos adquiridos nas aulas anteriores para desenvolver uma aplicação *multithread*. A construção da aplicação foi inspirada no código de Eugene Li [1], disponível no GitHub.

## Aplicação de bate-papo

Uma aplicação de bate-papo, também conhecida como *chat*, é um software que permite que usuários troquem mensagens em tempo real por uma rede de computadores. Essas aplicações são amplamente utilizadas para a comunicação entre indivíduos, havendo opções para conversas privadas, entre pequenos grupos, ou mesmo para interações em larga escala, como em salas de bate-papo públicas.

Um *chat* é uma conversa escrita **sincronizada**, entre dois ou mais indivíduos, seja em um *site* público que pode ser acessado com um nome de usuário, ou em um ambiente fechado (uma *sala*) para uso exclusivo dos membros (por exemplo, os alunos matriculados em um determinado componente curricular de um curso) aos quais é concedido acesso. Ele é composto por uma sequência de mensagens, tipicamente curtas, enviadas pelos participantes a partir de seus dispositivos.

O efeito proporcionado por um *chat* é a construção de uma troca escrita semelhante a um diálogo para uma peça, assemelhando-se em alguns aspectos à conversa falada (interação quase em tempo real, uso coloquial da linguagem, presença frequente de perguntas e

respostas, exclamações, frases inacabadas e breves). Exemplos conhecidos de *chats* são o ICQ (*I Seek You*), o IRC (*Internet Relay Chat*), o Facebook Messenger, o WhatsApp e o Discord (adaptado de [2]).

As aplicações de bate-papo são utilizadas para uma variedade de finalidades, incluindo comunicação pessoal, profissional, educacional entre outras.

A aplicação de *chat* a ser construída passo a passo neste material consiste em dois programas distintos: um **servidor** e um **cliente**. Esses programas permitirão que os usuários se conectem a um servidor de bate-papo e troquem mensagens em um ambiente de *chat* em tempo real. A aplicação final é uma versão simplificada de um sistema de bate-papo *multithread* em C.

## Pré-requisitos

Para concluir este tutorial com sucesso, espera-se o cumprimento dos seguintes requisitos de forma antecipada:

- Conhecimento básico sobre a linguagem de programação C.
- Conhecimento básico sobre o uso da biblioteca `pthread`.
- Disponibilização ou permissão para instalação de
  - um ambiente de desenvolvimento configurado; ou
  - um editor simples e um compilador C (preferencialmente `gcc`, pois os exemplos de aula fazem uso dele).
- Familiaridade com os conceitos abordados na disciplina.

## Projeto e Escopo da Aplicação

A aplicação funcionará através das duas ferramentas: um servidor de bate-papo e uma aplicação cliente. O programa será simples e servirá para demonstrar didaticamente os princípios básicos de comunicação entre cliente e servidor usando *sockets* em C.

A aplicação proposta será **multithreaded**, assim permitindo a comunicação simultânea de vários clientes.

O servidor deve ser uma aplicação de linha de comando que escutará em uma porta determinada aguardando a conexão de usuários do bate-papo (clientes). Esse servidor deve

- ser responsável por aceitar conexões de clientes e coordenar a comunicação entre eles.
- criar um *socket* para aguardar conexões de entrada dos clientes.
- aceitar conexões de clientes e criar uma nova *thread* para lidar com cada cliente, exceto
  - quando o cliente não se conectar corretamente
  - quando o número máximo de clientes for atingido
- manter uma lista de *sockets* de clientes ativos.
- adicionar a uma fila de mensagens toda mensagem enviada por um cliente.
- enviar as mensagens da fila para todos os clientes conectados (*broadcast*).
- possuir uma *thread* que gerencia novas conexões de clientes e outra que lida com mensagens recebidas.

O cliente é uma aplicação de linha de comando que permite aos usuários se conectarem ao servidor de bate-papo informando corretamente o *host* do servidor (nome ou IP) e a porta em que o serviço está sendo executado. Esse cliente deve

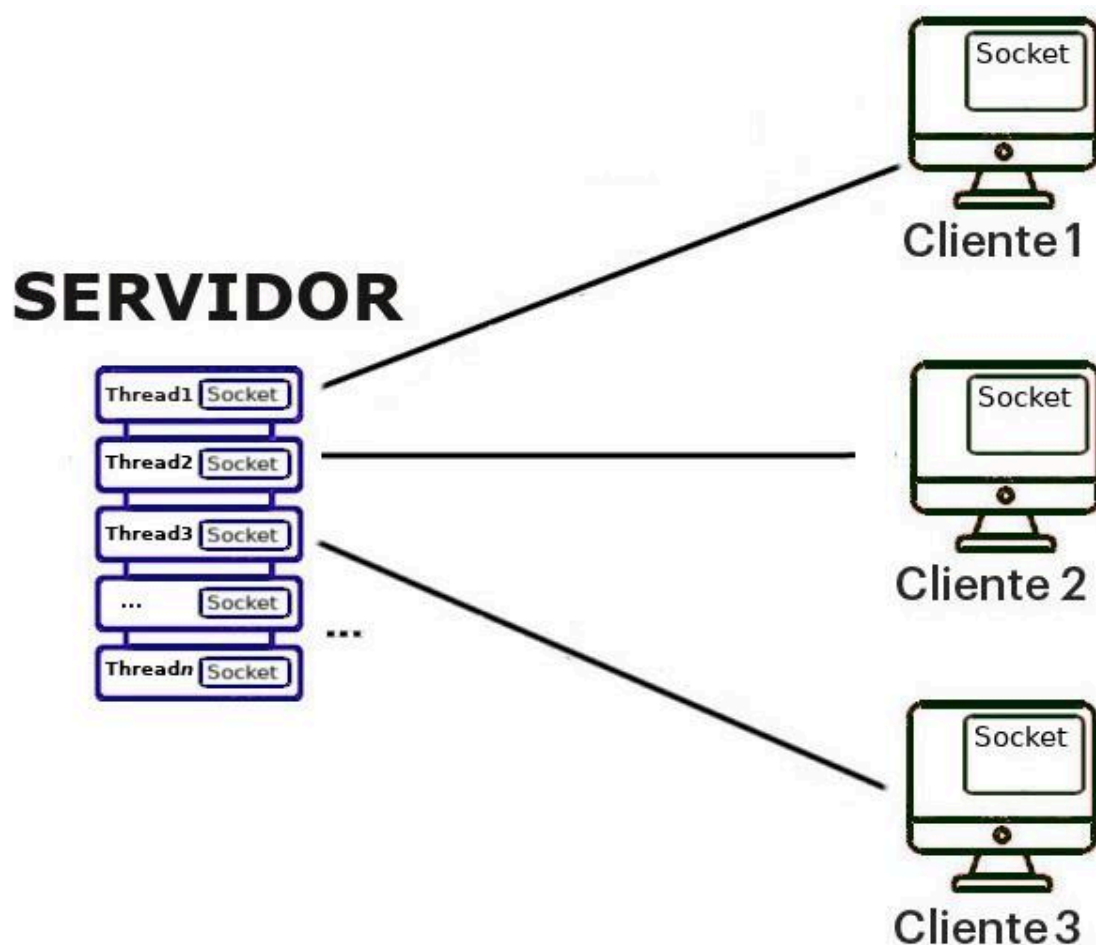
- aceitar três argumentos na linha de comando: nome de usuário, *host* do servidor e porta.
- criar um *socket* e tentar se conectar ao servidor.
- ser capaz de enviar mensagens de texto para o servidor, após a confirmação de conexão bem-sucedida.
- receber mensagens do servidor e exibi-las no *console*.
- incluir um mecanismo para sair controladamente do *chat* quando o usuário quiser (digitando `/sair [texto]`).

Até o momento da escrita deste tutorial, **não** fazem parte do escopo do projeto da aplicação de bate-papo as seguintes funcionalidades:

- opção de autenticação de usuários (clientes)
- definição de restrições no nome do usuário (por exemplo: nomes agressivos ou repetidos)
- moderação de mensagens
- alteração de nome de usuário
- envio de arquivos entre usuários
- opção de listagem de usuários ativos
- criação de salas de bate-papo
- (...)

**Obs.:** deseja-se futuramente incluir tais funcionalidades!

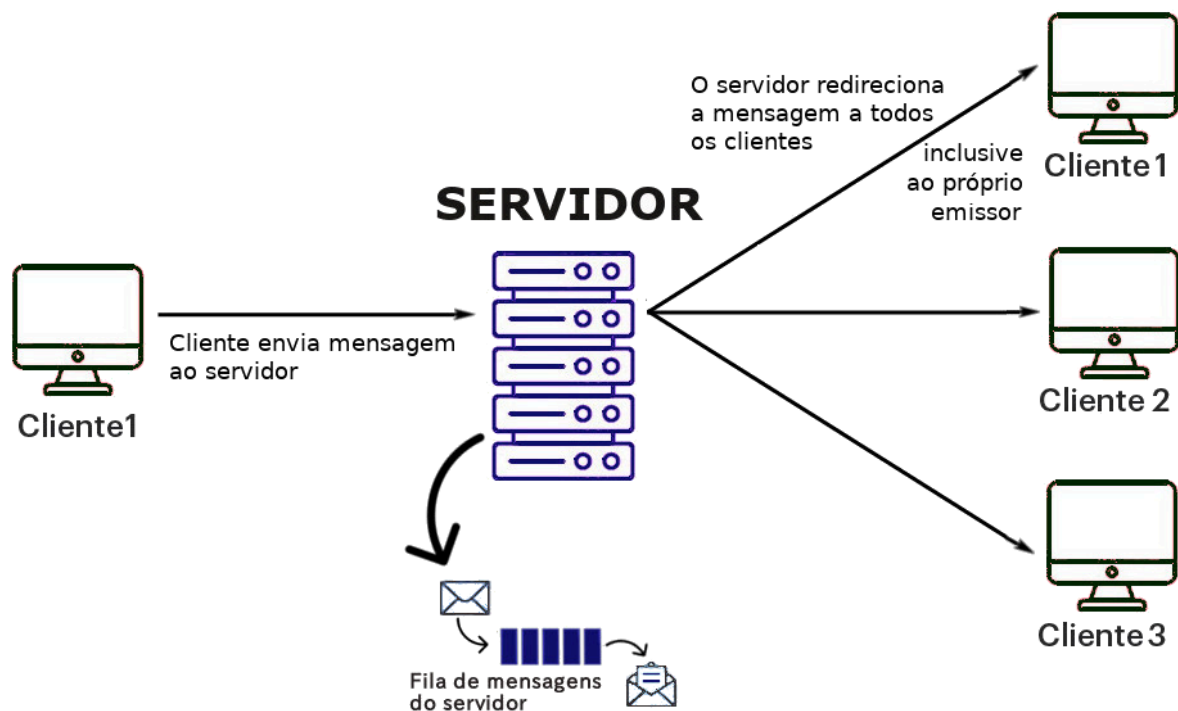
As Figuras 1 e 2 ilustram resumidamente a arquitetura da aplicação proposta. A Figura 1 mostra o servidor que escutará as conexões dos clientes. Quando um cliente se conectar ao *socket* do servidor, também chamado de *socket principal* (não presente explicitamente na figura), o servidor aceitará a conexão e criará uma nova *thread* para esse cliente. O *socket* principal será usado apenas para aceitar as conexões dos clientes, não para a comunicação direta com os clientes. Cada *thread* criada terá seu próprio *socket* dedicado para se comunicar com o cliente correspondente, e, portanto, terá seu próprio contexto de execução (veja na Figura 1). Assim, pode-se afirmar que haverá uma relação de "um cliente, uma *thread* e um *socket*" neste servidor, permitindo uma comunicação concorrente entre o próprio servidor e vários clientes ao mesmo tempo.



**Figura 1.** Ilustração da arquitetura da aplicação de bate-papo do ponto de vista da criação de *threads* e *sockets*. Fonte: elaborado pelo autor, adaptado de [3].

A Figura 2 traz a ideia do envio de mensagens na aplicação. Nela, o servidor possui, a título de exemplo, três clientes conectados. Quando um cliente decidir enviar uma mensagem (na Figura, o Cliente1), ele o fará para o servidor, que enviará esta mensagem para uma fila

de mensagens. As mensagens serão, uma a uma, enviadas a todos os clientes, inclusive ao próprio emissor, seguindo a ordem da fila.



**Figura 2.** Ilustração do envio de mensagens na aplicação de bate-papo. Fonte: elaborado pelo autor, adaptado de [3].

## Desenvolvimento da Aplicação

A construção da aplicação de bate-papo será dividida em algumas etapas.

### 1. Preparação do ambiente

Conforme comentado na [seção de pré-requisitos](#), é necessário instalar e configurar o ambiente de desenvolvimento de sua preferência. Caso seja necessário instalar o compilador `gcc` em sistemas Ubuntu (ou Debian), os comandos são:

```
> sudo apt-get update # atualiza a lista de pacotes disponíveis nos repositórios
```

```
> sudo apt-get install gcc # instala o compilador gcc
```

As bibliotecas necessárias para a aplicação funcionar já vêm instaladas por padrão no Ubuntu e em diversas outras distribuições Linux, então possivelmente nenhuma ação adicional seja necessária nesse sentido.

A próxima ação é criar o diretório da aplicação. Para isso, é necessário acessar um diretório em que haja permissão para escrita e criar um diretório. Exemplo:

```
> mkdir chat-application # cria o diretório chat-application dentro do diretório atual
```

Dentro do diretório `chat-application`, sugere-se já criar os arquivos do cliente (`cliente-chat-multithread.c`) e do servidor (`servidor-chat-multithread.c`):

```
> cd chat-application
```

```
> touch cliente-chat-multithread.c servidor-chat-multithread.c
```

## 2. Criação do Servidor

Esta seção descreve passo a passo a criação do servidor da aplicação de bate-papo.

### 2.1. Bibliotecas Necessárias

A criação do servidor se dará de forma incremental. Partir-se-á da criação do esqueleto do código e da inclusão das bibliotecas necessárias para que o servidor funcione:

```
#include <arpa/inet.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

int main() {

    printf("Ativando o servidor...\n");

    return 0;
}
```

Para cada código criado, uma boa prática é criar o arquivo executável e ver o seu funcionamento, por exemplo, digitando

```
> gcc nome-do-arquivo.c -o nome-do-arquivo-executavel -lpthread
> ./nome-do-arquivo [parâmetro(s)]
```

Neste primeiro código, tem-se como saída apenas a escrita de um texto padrão. No entanto, é importante observar quais são as bibliotecas necessárias para o funcionamento do servidor:

- `<arpa/inet.h>`: Essa biblioteca será necessária para garantir a correta conversão de valores entre a representação de *host* (que pode ser *little-endian* ou *big-endian*, dependendo da arquitetura do sistema) e a representação de rede (por padrão, *big-endian*). Em redes, endereços IP e números de porta precisam ser interpretados e transmitidos em uma ordem específica (*big-endian*) para garantir a compatibilidade entre diferentes sistemas. Serão utilizadas duas funções (`htons` e `htonl`) para realizar essas conversões de forma consistente.
- `<fcntl.h>`: Biblioteca necessária para configurar descritores de arquivo<sup>1</sup> (de *sockets*, que são tratados como arquivos).
- `<netinet/in.h>`: Necessário para tipos e estruturas relacionadas à manipulação de endereços IP e portas de rede que, por sua vez, serão necessários para trabalhar com *sockets*.
- `<pthread.h>`: Necessária para trabalhar com *threads*, incluindo a criação e gerenciamento de *threads* no servidor.
- `<stdio.h>`: Necessária para funções de entrada e saída padrão, como `fprintf` e `printf`.
- `<stdlib.h>`: Necessária para alocar e desalocar memória dinamicamente.
- `<string.h>`: Necessário para funções de manipulação, cópia e concatenação de *strings*, relacionadas às mensagens enviadas pelos usuários.
- `<unistd.h>`: Necessária para funções relacionadas ao sistema, como `close` e `read`, usadas para operações de *socket* e manipulação de arquivos.
- `<sys/socket.h>`: Necessária para manipulação de *sockets*, que serão necessários para estabelecer as conexões entre os usuários e o servidor, bem como para o envio e o recebimento de dados desses usuários.

## 2.2. Estruturas

O incremento seguinte consiste em criar estruturas (`structs`) necessárias para o funcionamento do servidor e fazer uso do argumento de linha de comando para receber o valor da porta:

---

<sup>1</sup> Abstração usada pelos sistemas operacionais para gerenciar operações em arquivos ou recursos de E/S, como *sockets* e dispositivos. Essas operações podem ser abertura, fechamento, leitura, escrita etc.

```

// Execute com ./executavel [porta], onde
// [porta] deve ser um número acima de 1023, por exemplo, 1234.
// Ela será a porta na qual o servidor escutará e receberá conexões.

#include <arpa/inet.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define TAMANHO_BUFFER 1024

// Struct que representa uma fila de mensagens usando um array de caracteres.
typedef struct {
    char *buffer[TAMANHO_BUFFER]; // Armazena as mensagens na fila.
    int inicio, cauda; // Índices para controlar a fila.
    int cheia, vazia; // Flags que indicam se a fila está cheia ou vazia.
    pthread_mutex_t *mutex; // Mutex para sincronização.
    pthread_cond_t *naoCheia, *naoVazia; // Variáveis de condição para
sincronização.
} Fila;

// Struct contendo dados importantes para o funcionamento do servidor.
typedef struct {
    fd_set descritoresLeituraServidor; // Conjunto de descritores de leitura para o
servidor.
    int socketFd; // Descritor do socket do servidor.
    int socketsClientes[TAMANHO_BUFFER]; // Array de descritores de sockets dos
clientes.
    int numClientes; // Número atual de clientes conectados.
    pthread_mutex_t *mutexListaClientes; // Mutex para lista de clientes.
    Fila *filaMensagens; // Fila de mensagens do servidor.
} DadosServidor;

// Struct simples para armazenar DadosServidor e o novo socket do cliente.
typedef struct {
    DadosServidor *dados;
    int socketCliente;
} DadosHandlerCliente;

int main(int argc, char *argv[]) {

```



```

// Declaração de uma estrutura para armazenar informações do servidor.
struct sockaddr_in serverAddr;

int porta;

// Verifica se foi fornecida uma porta na linha de comando.
if (argc == 2)
    porta = atoi(argv[1]);
else {
    fprintf(stderr, "Considere executar o programa da seguinte
forma:\n./executavel [porta]\n");
    exit(1);
}

printf("Ativando o servidor...\n");

return 0;
}

```

A primeira estrutura criada consiste em uma fila (`struct Fila`). Ela será utilizada para enfileirar mensagens. A proposta é que ela contenha os seguintes campos:

- **buffer**: Um *array* de ponteiros para caracteres que armazena as mensagens na fila.
- **inicio** e **cauda**: Índices que controlarão a fila, indicando onde ela começa e termina, respectivamente.
- **cheia** e **vazia**: Variáveis de controle que indicarão se a fila está cheia ou vazia, respectivamente.
- **mutex**: Um ponteiro para um *mutex* usado para sincronização entre *threads*. Assim, quando uma mensagem for enviada por um usuário do bate-papo, esta será inserida na fila de forma a garantir que mensagens de outros usuários não a sobreponham.
- **naoCheia** e **naoVazia**: Ponteiros para variáveis de condição<sup>2</sup> usadas para sinalização de fila não cheia e não vazia.

---

<sup>2</sup>Variáveis condicionais, representadas por `pthread_cond_t` no código, são usadas para coordenar a sincronização entre diferentes *threads* que estarão acessando a fila de mensagens. Elas são uma forma de permitir que as *threads* esperem até que uma condição específica seja atendida antes de continuar sua execução. No contexto do código proposto, as variáveis condicionais servirão para lidar com as situações em que a fila está cheia ou vazia, o que pode exigir que as *threads* aguardem até que a condição desejada seja alcançada.

A segunda estrutura serve para conter os dados do servidor (`struct DadosServidor`), o que inclui os seguintes campos:

- **descritoresLeituraServidor**: Um conjunto de descritores de leitura para o servidor.
- **socketFd**: O descritor do *socket* do servidor.
- **socketsClientes**: Um *array* de descritores de *sockets* dos clientes.
- **numClientes**: O número atual de clientes conectados.
- **mutexListaClientes**: Um ponteiro para um *mutex* usado para proteger a lista de clientes.
- **filaMensagens**: Um ponteiro para a fila de mensagens do servidor.

Foi definido que o *array* `socketsClientes` terá tamanho 1024. Isso foi feito criando uma constante simbólica `TAMANHO_BUFFER` associada ao valor 1024 através da diretiva de pré-processador `#define`. Assim, o máximo de clientes simultâneos que o servidor de bate-papo aceita é 1024. Da mesma forma, o tamanho do *buffer* que armazena mensagens na fila também foi definido pela constante `TAMANHO_BUFFER`, ou seja, tem tamanho 1024.

A terceira e última estrutura servirá para armazenar dados relacionados a um cliente (`struct DadosHandlerCliente`), e incluirá os seguintes campos:

- **dados**: Um ponteiro para a `struct DadosServidor`, que conterá os dados globais do servidor.
- **socketCliente**: O descritor do *socket* do cliente associado a esse *handler*<sup>3</sup>.

Também nesta etapa a proposta é declarar uma estrutura para armazenar informações do servidor. Isso é feito com `struct sockaddr_in serverAddr`, dentro da função `main`.

Essas estruturas serão importantes para organizar e gerenciar os dados da forma que a aplicação foi idealizada.

Ainda neste passo, estabeleceu-se também que o número da porta deve ser indicado na linha de comando, sendo atribuído à variável `porta`. Assim o servidor deve ser executado na forma abaixo, onde `porta` deve ser um número de porta maior que 1023, para evitar

---

<sup>3</sup>*Handler*, no contexto da aplicação aqui proposta, funciona como um manipulador das interações com o cliente.

conflitos com outros serviços conhecidos que são comumente associados a portas entre 0 e 1023 (*well-known ports*, como visto na [Aula 3 - Comunicação entre Processos](#)).

```
> ./nome-do-arquivo porta
```

### 2.3. Operações da Fila de Mensagens

Após definir as estruturas, pode-se criar as operações que as manipularão, a começar pela fila de mensagens. São necessárias operações para criação e destruição da fila, bem como de enfileiramento e desenfileiramento de mensagens. A terceira versão do código fica assim:

```
// Execute com ./executavel [porta], onde
// [porta] deve ser um número acima de 1023, por exemplo, 1234.
// Ela será a porta na qual o servidor escutará e receberá conexões.

#include <arpa/inet.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define TAMANHO_BUFFER 1024

// Struct que representa uma fila de mensagens usando um array de caracteres.
typedef struct {
    char *buffer[TAMANHO_BUFFER]; // Armazena as mensagens na fila.
    int inicio, cauda; // Índices para controlar a fila.
    int cheia, vazia; // Flags que indicam se a fila está cheia ou vazia.
    pthread_mutex_t *mutex; // Mutex para sincronização.
    pthread_cond_t *naoCheia, *naoVazia; // Variáveis de condição para
    sincronização.
} Fila;

// Struct contendo dados importantes para o funcionamento do servidor.
typedef struct {
    fd_set descritoresLeituraServidor; // Conjunto de descritores de leitura para o
    servidor.
    int socketFd; // Descritor do socket do servidor.
    int socketsClientes[TAMANHO_BUFFER]; // Array de descritores de sockets dos
    clientes.
```

```

    int numClientes; // Número atual de clientes conectados.
    pthread_mutex_t *mutexListaClientes; // Mutex para lista de clientes.
    Fila *filaMensagens; // Fila de mensagens do servidor.
} DadosServidor;

// Struct simples para armazenar DadosServidor e o novo socket do cliente.
typedef struct {
    DadosServidor *dados;
    int socketCliente;
} DadosHandlerCliente;

// Inicializa a fila de mensagens do servidor.
Fila* inicializarFila() {
    // Aloca memória para a estrutura de fila.
    Fila *fila = (Fila *)malloc(sizeof(Fila));
    if (fila == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }

    // Inicializa os valores da fila.
    fila->vazia = 1;
    fila->cheia = fila->inicio = fila->cauda = 0;

    // Aloca e inicializa um mutex para proteger operações na fila.
    fila->mutex = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
    if (fila->mutex == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_init(fila->mutex, NULL);

    // Aloca e inicializa variáveis condicionais para sinalização de fila não cheia
    e não vazia.
    fila->naoCheia = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
    if (fila->naoCheia == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }
    pthread_cond_init(fila->naoCheia, NULL);

    fila->naoVazia = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
    if (fila->naoVazia == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }

```

```

    }

    pthread_cond_init(&fila->naoVazia, NULL);

    return fila;
}

// Insere uma mensagem no final da fila de mensagens do servidor.
void enfileirar(Fila *fila, char* mensagem) {
    // Armazena a mensagem no buffer da fila na posição da cauda.
    fila->buffer[fila->cauda] = mensagem;

    // Move a posição da cauda para o próximo índice no buffer circular.
    fila->cauda++;

    // Verifica se a cauda atingiu o final do buffer circular e, se sim, volta para
    o início.
    if (fila->cauda == TAMANHO_BUFFER)
        fila->cauda = 0;

    // Se a cauda alcançou a cabeça da fila, significa que a fila está cheia.
    if (fila->cauda == fila->inicio)
        fila->cheia = 1;

    // Marca a fila como não vazia.
    fila->vazia = 0;
}

// Remove e retorna a mensagem mais antiga da fila de mensagens do servidor.
char* desenfileirar(Fila *fila) {
    // Obtém a mensagem mais antiga da fila (na posição da cabeça).
    char* mensagem = fila->buffer[fila->inicio];

    // Move a cabeça da fila para a próxima posição.
    fila->inicio++;

    // Verifica se a cabeça atingiu o final da fila e a reinicia se necessário.
    if (fila->inicio == TAMANHO_BUFFER)
        fila->inicio = 0;

    // Verifica se a fila está agora vazia após a remoção.
    if (fila->inicio == fila->cauda)
        fila->vazia = 1;

    // Marca a fila como não cheia, pois um elemento foi removido.
    fila->cheia = 0;
}

```

```

        // Retorna a mensagem removida.
        return mensagem;
    }

    // Libera a memória associada à fila.
    void destruirFila(Fila *fila) {
        // Destrói o mutex associado à fila.
        pthread_mutex_destroy(&fila->mutex);

        // Destrói as variáveis de condição associadas à fila (para controle de cheio e
        // vazio).
        pthread_cond_destroy(&fila->naoCheia);
        pthread_cond_destroy(&fila->naoVazia);

        // Libera a memória alocada para o mutex.
        free(fila->mutex);

        // Libera a memória alocada para as variáveis de condição.
        free(fila->naoCheia);
        free(fila->naoVazia);

        // Libera a memória alocada para a própria estrutura de fila.
        free(fila);
    }

    int main(int argc, char *argv[]) {
        // Declaração de uma estrutura para armazenar informações do servidor.
        struct sockaddr_in serverAddr;

        int porta;

        // Verifica se foi fornecida uma porta na linha de comando.
        if (argc == 2)
            porta = atoi(argv[1]);
        else {
            fprintf(stderr, "Considere executar o programa da seguinte
            forma:\n./executavel [porta]\n");
            exit(1);
        }

        printf("Ativando o servidor...\n");

        return 0;
    }

```

No código, é possível identificar as quatro operações citadas:

- **Fila\* inicializarFila():** Função responsável por inicializar a estrutura, alocando memória para a fila, inicializando seus atributos, mutexes e variáveis de condição necessárias para sincronização em um ambiente *multithreading*.
- **void enfileirar(Fila \*fila, char\* mensagem):** Função que será chamada quando um cliente enviar uma mensagem. Ela inserirá uma mensagem (no fim) da fila. A mensagem será armazenada no *buffer* da fila na posição indicada pela cauda. A função também tem como propósito atualizar os índices da fila e verificar se ela está cheia.
- **char\* desenfileirar(Fila \*fila):** Função que removerá e retornará a mensagem mais antiga da fila. Ela será chamada por um *handler de mensagens* para obter a próxima mensagem a ser enviada aos clientes.
- **void destruirFila(Fila \*fila):** Função que será responsável por liberar a memória associada à fila quando ela não for mais necessária.

#### 2.4. Threads e Manipulação de Clientes

Em relação à lista de clientes, faz sentido implementar uma função para remover os usuários do bate-papo quando eles desconectarem. Isso será feito pela função `removerCliente`:

```
void removerCliente(DadosServidor *dados, int socketCliente) {
    // Obtém o mutex para acessar com segurança a lista de clientes ativos.
    pthread_mutex_lock(dados->mutexListaClientes);

    // Procura o socket do cliente na lista de clientes ativos.
    for (int i = 0; i < TAMANHO_BUFFER; i++) {
        if (dados->socketsClientes[i] == socketCliente) {
            dados->socketsClientes[i] = 0;
            close(socketCliente);
            dados->numClientes--;

            // Sai do loop depois de encontrar o socket.
            i = TAMANHO_BUFFER;
        }
    }

    // Libera o mutex para permitir outros acessos concorrentes à lista de clientes
    ativos.
    pthread_mutex_unlock(dados->mutexListaClientes);
}
```

Nessa etapa ainda são necessárias as seguintes funções:

- **Handler do cliente:** Função responsável por lidar com as mensagens recebidas do cliente. Se o cliente enviar a mensagem `"/sair"`, ela chamará `removerCliente` para desconectar o cliente.
- **Criação de threads para novos clientes:** Através da função `novoHandlerCliente`, a aplicação criará uma nova *thread* para cada novo cliente, onde será feita a conexão do cliente com o servidor por meio da adição do *socket* do cliente à lista de *sockets* de clientes ativos e será chamada a função `handlerCliente` para lidar com as mensagens recebidas por esse cliente.
- **Handler de Mensagens:** Função que enviará mensagens em *broadcast* para todos os clientes ativos.
- **Associação de socket:** A função `associarSocket` configurará e associará o *socket* do servidor a um endereço IP e porta específicos.

O código resultante dessas ações fica da seguinte forma:

```
// Execute com ./executavel [porta], onde
// [porta] deve ser um número acima de 1023, por exemplo, 1234.
// Ela será a porta na qual o servidor escutará e receberá conexões.

#include <arpa/inet.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define TAMANHO_BUFFER 1024

// Struct que representa uma fila de mensagens usando um array de caracteres.
typedef struct {
    char *buffer[TAMANHO_BUFFER]; // Armazena as mensagens na fila.
    int inicio, cauda; // Índices para controlar a fila.
    int cheia, vazia; // Flags que indicam se a fila está cheia ou vazia.
    pthread_mutex_t *mutex; // Mutex para sincronização.
    pthread_cond_t *naoCheia, *naoVazia; // Variáveis de condição para
    sincronização.
} Fila;
```



```

// Struct contendo dados importantes para o funcionamento do servidor.
typedef struct {
    fd_set descritoresLeituraServidor; // Conjunto de descritores de leitura para o
servidor.

    int socketFd; // Descritor do socket do servidor.

    int socketsClientes[TAMANHO_BUFFER]; // Array de descritores de sockets dos
clientes.

    int numClientes; // Número atual de clientes conectados.

    pthread_mutex_t *mutexListaClientes; // Mutex para lista de clientes.

    Fila *filaMensagens; // Fila de mensagens do servidor.
} DadosServidor;

// Struct simples para armazenar DadosServidor e o novo socket do cliente.
typedef struct {
    DadosServidor *dados;

    int socketCliente;
} DadosHandlerCliente;

// Inicializa a fila de mensagens do servidor.
Fila* inicializarFila() {
    // Aloca memória para a estrutura de fila.
    Fila *fila = (Fila *)malloc(sizeof(Fila));
    if (fila == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }

    // Inicializa os valores da fila.
    fila->vazia = 1;
    fila->cheia = fila->inicio = fila->cauda = 0;

    // Aloca e inicializa um mutex para proteger operações na fila.
    fila->mutex = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
    if (fila->mutex == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_init(fila->mutex, NULL);

    // Aloca e inicializa variáveis condicionais para sinalização de fila não cheia
e não vazia.
    fila->naoCheia = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
    if (fila->naoCheia == NULL) {
        perror("Não foi possível alocar mais memória!");
    }
}

```

```

        exit(EXIT_FAILURE);
    }
    pthread_cond_init(&fila->naoCheia, NULL);

    fila->naoVazia = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
    if (fila->naoVazia == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }
    pthread_cond_init(&fila->naoVazia, NULL);

    return fila;
}

// Insere uma mensagem no final da fila de mensagens do servidor.
void enqueue(Fila *fila, char* mensagem) {
    // Armazena a mensagem no buffer da fila na posição da cauda.
    fila->buffer[fila->cauda] = mensagem;

    // Move a posição da cauda para o próximo índice no buffer circular.
    fila->cauda++;

    // Verifica se a cauda atingiu o final do buffer circular e, se sim, volta para
    // o início.
    if (fila->cauda == TAMANHO_BUFFER)
        fila->cauda = 0;

    // Se a cauda alcançou a cabeça da fila, significa que a fila está cheia.
    if (fila->cauda == fila->inicio)
        fila->cheia = 1;

    // Marca a fila como não vazia.
    fila->vazia = 0;
}

// Remove e retorna a mensagem mais antiga da fila de mensagens do servidor.
char* dequeue(Fila *fila) {
    // Obtém a mensagem mais antiga da fila (na posição da cabeça).
    char* mensagem = fila->buffer[fila->inicio];

    // Move a cabeça da fila para a próxima posição.
    fila->inicio++;

    // Verifica se a cabeça atingiu o final da fila e a reinicia se necessário.
    if (fila->inicio == TAMANHO_BUFFER)

```

```

        fila->inicio = 0;

// Verifica se a fila está vazia após a remoção.
if (fila->inicio == fila->cauda)
    fila->vazia = 1;

// Marca a fila como não cheia, pois um elemento foi removido.
fila->cheia = 0;

// Retorna a mensagem removida.
return mensagem;
}

// Libera a memória associada à fila.
void destruirFila(Fila *fila) {
    // Destrói o mutex associado à fila.
    pthread_mutex_destroy(fila->mutex);

    // Destrói as variáveis de condição associadas à fila (para controle de cheio e
    vazio).
    pthread_cond_destroy(fila->naoCheia);
    pthread_cond_destroy(fila->naoVazia);

    // Libera a memória alocada para o mutex.
    free(fila->mutex);

    // Libera a memória alocada para as variáveis de condição.
    free(fila->naoCheia);
    free(fila->naoVazia);

    // Libera a memória alocada para a própria estrutura de fila.
    free(fila);
}

// Remove o socket da lista de sockets de cliente ativos e fecha-o.
void removerCliente(DadosServidor *dados, int socketCliente) {
    // Obtém o mutex para acessar com segurança a lista de clientes ativos.
    pthread_mutex_lock(dados->mutexListaClientes);

    // Procura o socket do cliente na lista de clientes ativos.
    for (int i = 0; i < TAMANHO_BUFFER; i++) {
        if (dados->socketsClientes[i] == socketCliente) {
            dados->socketsClientes[i] = 0;
            close(socketCliente);
            dados->numClientes--;
        }
    }
}

```

```

        // Sai do loop depois de encontrar o socket.
        i = TAMANHO_BUFFER;
    }
}

// Libera o mutex para permitir outros acessos concorrentes à lista de clientes
ativos.
pthread_mutex_unlock(dados->mutexListaClientes);
}

// Handler do cliente que lida com mensagens recebidas do cliente.
void *handlerCliente(void *dhc) {
    // Converte o ponteiro genérico de dados de handler do cliente de volta para o
    tipo DadosHandlerCliente.
    DadosHandlerCliente *dadosHandler = (DadosHandlerCliente *)dhc;
    DadosServidor *dadosServidor = (DadosServidor *)dadosHandler->dados;

    // Obtém a fila de mensagens e o socket do cliente deste handler.
    Fila *fila = dadosServidor->filaMensagens;
    int socketCliente = dadosHandler->socketCliente;

    char bufferMensagem[TAMANHO_BUFFER];
    while (1) {
        // Lê mensagens recebidas do cliente e armazena no buffer.
        int numBytesLidos = read(socketCliente, bufferMensagem, TAMANHO_BUFFER -
1);

        bufferMensagem[numBytesLidos] = '\0';

        // Se o cliente enviou "/sair\n", remove-o da lista de clientes e fecha o
        socket dele.
        if (!strcmp(bufferMensagem, "/sair\n", 6) || !strcmp(bufferMensagem,
"/sair ", 6)) {
            fprintf(stderr, "Cliente desconectado. Socket: %d.\n", socketCliente);
            removerCliente(dadosServidor, socketCliente);
            return NULL; // Encerra a thread do cliente.
        }
        else {
            // Espera até que a fila não esteja cheia antes de inserir uma
            mensagem.

            while (fila->cheia) {
                pthread_cond_wait(fila->naoCheia, fila->mutex);
            }

```

```

        // Obtém lock na fila, insere a mensagem na fila, libera o lock e
        sinaliza a variável condicional.
        pthread_mutex_lock(fila->mutex);
        fprintf(stderr, "Inserindo mensagem na fila: %s", bufferMensagem);
        enfileirar(fila, bufferMensagem);
        pthread_mutex_unlock(fila->mutex);
        pthread_cond_signal(fila->naoVazia); // Notifica o handler de mensagens
        que há uma nova mensagem na fila.
    }
}

// Thread para lidar com novas conexões. Adiciona o fd do cliente à lista de fds do
cliente e gera uma nova thread handlerCliente para ele.
void *novoHandlerCliente(void *dados) {
    // Converte o ponteiro genérico de dados de volta para o tipo DadosServidor.
    DadosServidor *dadosServidor = (DadosServidor *)dados;

    while(1) {
        // Aceita uma nova conexão de cliente e obtém o socket do cliente.
        int socketCliente = accept(dadosServidor->socketFd, NULL, NULL);

        // Verifica se a conexão foi aceita com sucesso.
        if(socketCliente > 0) {
            fprintf(stderr, "Servidor aceitou novo cliente. Socket: %d.\n",
socketCliente);

            // Obtém lock na lista de clientes.
            pthread_mutex_lock(dadosServidor->mutexListaClientes);

            // Verifica se o número de clientes ativos ainda não atingiu o limite.
            if (dadosServidor->numClientes < TAMANHO_BUFFER) {
                // Adiciona o novo cliente à lista de sockets de clientes ativos.
                for (int i = 0; i < TAMANHO_BUFFER; i++) {
                    // Encontra uma posição vazia na lista de sockets de clientes.
                    if (!FD_ISSET(dadosServidor->socketsClientes[i],
&(dadosServidor->descritoresLeituraServidor))) {
                        dadosServidor->socketsClientes[i] = socketCliente;
                        i = TAMANHO_BUFFER; // Sai do loop.
                    }
                }
            }

            // Adiciona o novo socket à lista de descritores de leitura do
servidor.

```

```

        FD_SET(socketCliente,
&(dadosServidor->descritoresLeituraServidor));

        // Gera uma nova thread para lidar com mensagens do cliente.
        DadosHandlerCliente dhc;
        dhc.socketCliente = socketCliente;
        dhc.dados = dadosServidor;

        pthread_t threadCliente;
        if (!pthread_create(&threadCliente, NULL, (void *)&handlerCliente,
(void *)&dhc)) {

            // Incrementa o contador de clientes ativos e exibe uma
mensagem.

            dadosServidor->numClientes++;
            fprintf(stderr, "Cliente entrou no chat. Socket: %d.\n",
socketCliente);
        }
        else
            close(socketCliente); // Fecha o socket em caso de falha na
criação da thread.
    }

    // Libera o lock na lista de clientes.
    pthread_mutex_unlock(dadosServidor->mutexListaClientes);
}

}

}

// Handler de mensagens que espera até que a fila de mensagens não esteja vazia e
envia em broadcast para os clientes.
void *handlerMensagens(void *dados) {
    // Converte os dados genéricos para o tipo de estrutura DadosServidor.
    DadosServidor *dadosServidor = (DadosServidor *)dados;

    // Obtém a referência para a fila de mensagens do servidor.
    Fila *fila = dadosServidor->filaMensagens;

    // Obtém a referência para o array de sockets dos clientes.
    int *socketsClientes = dadosServidor->socketsClientes;

    while(1) {
        // Obtém um bloqueio no mutex da fila.
        pthread_mutex_lock(fila->mutex);

        // Aguarda até que a fila de mensagens não esteja vazia.

```

```

while(fila->vazia) {
    pthread_cond_wait(fila->naoVazia, fila->mutex);
}

// Remove a mensagem mais antiga da fila.
char* mensagem = desenfileirar(fila);

// Libera o bloqueio do mutex da fila e sinaliza que não está mais cheia.
pthread_mutex_unlock(fila->mutex);
pthread_cond_signal(fila->naoCheia);

// Mensagem em broadcast para todos os clientes conectados.
fprintf(stderr, "Mensagem em broadcast: %s", mensagem);

// Envia a mensagem para todos os clientes ativos.
for(int i = 0; i < dadosServidor->numClientes; i++) {
    int socket = socketsClientes[i];
    if (socket != 0 && write(socket, mensagem, TAMANHO_BUFFER - 1) == -1)
        perror("Falha na escrita do socket: ");
}
}

// Configura e associa o socket a um endereço IP e porta específicos.
void associarSocket(struct sockaddr_in *serverAddr, int socketFd, long port) {
    // Limpa a estrutura serverAddr para garantir que não haja dados residuais.
    memset(serverAddr, 0, sizeof(*serverAddr));

    // Define o tipo de endereço como IPv4.
    serverAddr->sin_family = AF_INET;

    // Configura o endereço IP para INADDR_ANY, que significa "escutar em todas as
    interfaces de rede".
    serverAddr->sin_addr.s_addr = htonl(INADDR_ANY);

    // Configura a porta do servidor para a porta especificada.
    serverAddr->sin_port = htons(port);

    // Tenta associar o socket ao endereço e porta especificados.
    if (bind(socketFd, (struct sockaddr *)serverAddr, sizeof(struct sockaddr_in))
    == -1) {
        perror("Falha ao associar o socket: ");
        exit(1);
    }
}

```

```

int main(int argc, char *argv[]) {
    // Declaração de uma estrutura para armazenar informações do servidor.
    struct sockaddr_in serverAddr;

    int porta;

    // Declaração do descritor de arquivo do socket.
    int socketFd;

    // Verifica se foi fornecida uma porta na linha de comando.
    if (argc == 2)
        porta = atoi(argv[1]);
    else {
        fprintf(stderr, "Considere executar o programa da seguinte
forma:\n./executavel [porta]\n");
        exit(1);
    }

    printf("Ativando o servidor...\n");

    // Cria um novo socket usando o protocolo IPv4 e TCP.
    if ((socketFd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Falha na criação do socket.");
        exit(1);
    }

    // Associa o socket ao endereço e porta especificados.
    associarSocket(&serverAddr, socketFd, porta);

    // Inicia o servidor para ouvir novas conexões na porta especificada.
    if (listen(socketFd, 1) == -1) {
        perror("Falha na escuta: ");
        exit(1);
    }

    return 0;
}

```

É importante observar que dentro da função `novoHandlerCliente` há um *loop* infinito que aguardará novas conexões de clientes usando a função `accept`. Esta função bloqueará a execução do servidor até que um cliente se conecte. Isso significa que o servidor entrará em um estado de *espera ocupada*, ou *espera ativa* (expressão explicada na [Aula 5 – Sincronização](#)), período durante o qual o servidor não executará outras tarefas,



pois estará ocupado verificando constantemente se há conexões pendentes. A espera ocupada também foi utilizada nos *handlers* do cliente e de mensagens.

Embora a abordagem de espera ocupada seja simples de implementar (motivo pelo qual ela foi adotada na construção da aplicação de *chat* aqui proposta), ela não é considerada eficiente, especialmente se a aplicação receber muitas conexões. A citada verificação contínua realizada por parte do servidor consome recursos da CPU, o que não é desejável.

Voltando à análise do código obtido até o momento, é possível perceber que mesmo com as novas alterações ele ainda não inicia de fato o servidor do *chat*. Isso ocorre porque a função responsável por iniciar o *chat*, que deve chamar outras funções e configurar o ambiente para que a comunicação entre o servidor e os clientes seja estabelecida, ainda não foi implementada. Essa função será chamada `iniciarChat`.

## 2.5. Integração e Inicialização do *Chat*

A execução efetiva do *chat* não ocorrerá sem a implementação da função `iniciarChat`. É necessário criar essa função para orquestrar a inicialização do servidor e o tratamento das conexões dos clientes. Essa função deverá chamar outras funções apropriadas para configurar o servidor, inicializar a fila de mensagens e iniciar as *threads* para lidar com as conexões dos clientes.

O código da função `iniciarChat` deve ser como segue:

```
// Inicia as threads do servidor para lidar com conexões e mensagens.
void iniciarChat(int socketFd) {
    // Cria uma estrutura para armazenar os dados do servidor.
    DadosServidor dados;
    memset(&dados, 0, sizeof dados);

    // Inicializa o número de clientes como zero.
    dados.numClientes = 0;

    // Configura o descritor de socket do servidor.
    dados.socketFd = socketFd;

    // Inicializa a fila de mensagens do servidor.
    dados.filaMensagens = inicializarFila();

    // Aloca memória e inicializa o mutex para a lista de clientes.
    dados.mutexListaClientes = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
```

```

pthread_mutex_init(&dados.mutexListaClientes, NULL);

// Inicia a thread para lidar com novas conexões de clientes.
pthread_t threadConexao;
if (!pthread_create(&threadConexao, NULL, (void *)&novoHandlerCliente, (void
*)&dados)) {
    fprintf(stderr, "Handler de conexões iniciado.\n");
}

// Configura o conjunto de descritores de leitura do servidor.
FD_ZERO(&(dados.descritoresLeituraServidor));
FD_SET(socketFd, &(dados.descritoresLeituraServidor));

// Inicia a thread para lidar com mensagens recebidas.
pthread_t threadMensagens;
if (!pthread_create(&threadMensagens, NULL, (void *)&handlerMensagens, (void
*)&dados)) {
    fprintf(stderr, "Handler de mensagens iniciado.\n");
}

// Aguarda a conclusão das threads de conexão e mensagens.
pthread_join(threadConexao, NULL);
pthread_join(threadMensagens, NULL);

// Libera a memória utilizada pela fila de mensagens.
destruirFila(dados.filaMensagens);

// Destroi o mutex da lista de clientes e libera a memória associada.
pthread_mutex_destroy(&dados.mutexListaClientes);
free(dados.mutexListaClientes);
}

```

É possível perceber que o código precisa de *threads* para permitir o tratamento concorrente de diferentes aspectos do servidor. A primeira *thread* (`threadConexao`) é necessária para lidar com novas conexões de clientes. Cada vez que um cliente se conectar ao servidor, a função `novoHandlerCliente` será chamada em uma nova *thread*, permitindo que o servidor aceite conexões concorrentemente, sem bloquear o fluxo principal. Isso é importante para garantir que o servidor possa lidar com múltiplos clientes simultaneamente.

A segunda *thread* (`threadMensagens`) é necessária para lidar com o envio de mensagens a partir da fila de mensagens. Essa *thread* permitirá que o servidor envie mensagens para os clientes de forma assíncrona, sem bloquear o servidor principal. Isso será importante

para garantir que o servidor possa enviar mensagens para os clientes de forma eficiente e sem atrasos. Utilizando *threads*, essas operações podem ser paralelizadas, o que é essencial para um servidor eficiente. A função `pthread_join` é utilizada para aguardar a conclusão das *threads* de conexão e mensagens, garantindo que o programa principal esperará até que essas *threads* terminem antes de encerrar a execução. Isso é fundamental para garantir que o programa principal não termine antes que as operações de tratamento estejam completas.

Além disso, será necessário incluir a chamada à função `iniciarChat` na função `main`:

```
iniciarChat(socketFd);
```

Bem como chamar a instrução de fechamento do *socket* após a execução:

```
close(socketFd);
```

### 3. Criação do Cliente

Esta seção descreve passo a passo a criação da aplicação utilizada pelos clientes do bate-papo (os usuários).

#### 3.1. Bibliotecas Necessárias

A criação do cliente se dará de forma incremental. Partir-se-á da criação do esqueleto do código e da inclusão das bibliotecas necessárias para que a aplicação cliente funcione:

```
#include <arpa/inet.h>
#include <fcntl.h>
#include <netdb.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <time.h>
#include <unistd.h>

int main() {

    printf("Ativando o cliente...\n");

    return 0;
```

}

Neste primeiro código, tem-se como saída apenas a escrita de um texto padrão. No entanto, assim como foi feito com o servidor, é importante observar quais são as bibliotecas necessárias para o funcionamento do cliente:

- **<arpa/inet.h>**: Será usada para funções relacionadas à manipulação de endereços IP.
- **<fcntl.h>**: Será utilizada para operações de controle de descritores de arquivo, principalmente para definir descritores de arquivo como não bloqueantes.
- **<netdb.h>**: Será usada para operações relacionadas à resolução de nomes de *host*, como a função `gethostbyname`, que obtém informações do *host* a partir de seu nome.
- **<netinet/in.h>**: Contém definições de estruturas para operações de rede, úteis para representar endereços de *socket*.
- **<signal.h>**: Lida com sinais, permitindo a definição de manipuladores para sinais específicos, como o manipulador de interrupção (`SIGINT`), chamado quando um usuário fecha a execução da aplicação digitando `CTRL+C`.
- **<stdio.h>**: Biblioteca de entrada e saída padrão, que contém `printf` e `fprintf`.
- **<stdlib.h>**: Utilizada para alocação de memória (`malloc`), entre outras operações.
- **<string.h>**: Útil para manipulação de *strings*, por exemplo, para formatar as mensagens dos usuários.
- **<sys/types.h>** e **<sys/socket.h>**: Contém definições de tipos e funções relacionadas a *sockets*, necessários para estabelecer a comunicação entre os processos cliente e servidor pela rede.
- **<time.h>**: Será usada para trabalhar com funções relacionadas ao tempo, como `time` e `strftime`. Elas serão úteis para que a mensagem enviada pelos usuários seja formatada com a hora de envio.
- **<unistd.h>**: Será usada para funcionalidades relacionadas a operações de entrada/saída e controle de execução no programa. Inclui funções para manipulação de descritores de arquivo, permitindo a comunicação com o servidor por meio do *socket*. Além disso, fornece funções para encerrar conexões e para monitorar múltiplos descritores de arquivo, facilitando a recepção de mensagens de forma assíncrona.

Note que para o código do cliente não é necessário incluir `pthread.h`, pois não há uso de *threads* em seu código. No cliente, a execução ocorrerá em um único fluxo (*thread*) de controle, e com isso não há necessidade de incluir a biblioteca de *threads*.

### 3.2. Constantes e Variáveis

O incremento seguinte consiste em declarar constantes e variáveis necessárias para o funcionamento do cliente e fazer uso do argumento de linha de comando para receber os parâmetros necessários para estabelecer a conexão com o servidor.

Utilizar-se-á a variável global `static int socketFd` como um descritor do *socket* do cliente. O descritor de arquivo (*file descriptor*) é um identificador numérico usado para acessar o *socket* em operações de leitura, escrita e gerenciamento. Nesse caso, `socketFd` está sendo usado para manter a referência ao *socket* de comunicação que o cliente estabelece com o servidor. Através desse descritor, o cliente pode enviar e receber mensagens do servidor, mantendo a conexão ativa durante a execução do programa.

Também definiu-se uma constante simbólica `TAMANHO_BUFFER` associada ao valor 1024 através da diretiva de pré-processador `#define`. Assim, em concordância com o que foi feito no servidor, o tamanho do *buffer* que armazena mensagens na fila terá tamanho 1024.

Neste passo, a função `main` passa a ser escrita da seguinte forma:

```
int main(int argc, char *argv[]) {
    char *nome;
    struct sockaddr_in enderecoServidor;
    struct hostent *host;
    long porta;

    if (argc != 4) {
        fprintf(stderr, "Considere executar o programa da seguinte
forma:\n./executavel [usuario] [host] [porta]\n");
        exit(1);
    }

    nome = argv[1];
    // Obtém informações sobre o host (nome do servidor) a partir do DNS.
    host = gethostbyname(argv[2]);
    porta = atoi(argv[3]);

    printf("Ativando o cliente...\n");
```

```

    return 0;
}

```

Assim, observa-se que o cliente deve ser executado sempre na forma `./executavel [usuário] [host] [porta]`, onde

- `usuário` deve ser o nome do usuário a acessar o *chat*;
- `host` deve ser o nome do servidor ou o seu endereço IP;
- `porta` deve ser o número da porta na qual o servidor está escutando.

Ainda devem ser declaradas duas estruturas:

- **`struct sockaddr_in enderecoServidor`**: Estrutura utilizada para armazenar informações sobre o endereço do servidor ao qual o cliente deseja se conectar. Essa estrutura do tipo `sockaddr_in` é específica para comunicações através do protocolo IPv4.
- **`struct hostent *host`**: Ponteiro para uma estrutura que contém informações sobre um *host*, como seu nome e endereço IP. A função `gethostbyname` será usada para obter informações sobre o *host* a partir de seu nome. A estrutura `hostent` armazenará essas informações.

### 3.3. Configuração e Conexão com o Socket

Essa etapa consiste em estabelecer a configuração do `socket` e a comunicação com o servidor. Assim, a função abaixo deve ser implementada:

```

void configurarEConectar(struct sockaddr_in *enderecoServidor, struct hostent
*host, int socketFd, long porta) {
    memset(enderecoServidor, 0, sizeof(enderecoServidor));
    enderecoServidor->sin_family = AF_INET;
    enderecoServidor->sin_addr = *((struct in_addr *)host->h_addr_list[0]);
    enderecoServidor->sin_port = htons(porta);

    // Tenta estabelecer uma conexão com o servidor.
    if (connect(socketFd, (struct sockaddr *) enderecoServidor, sizeof(struct
sockaddr)) < 0) {
        perror("Não foi possível conectar ao servidor.");
        exit(1);
    }
}

```

A função `configurarEConectar` desempenha um papel importante na preparação do cliente para se conectar ao servidor. Ela limpa a estrutura de endereço do servidor (`enderecoServidor`) para evitar dados residuais. Em seguida, configura o tipo de endereço para IPv4 e obtém o endereço IP do servidor usando as informações do *host*. Posteriormente, configura a porta do servidor para a porta especificada. Finalmente, tenta estabelecer uma conexão com o servidor utilizando o descritor de arquivo do *socket*, a estrutura de endereço do servidor e seu tamanho. Se a conexão falha, exibe uma mensagem de erro e encerra a execução do programa.

Essa função deve ser chamada na função principal, pois é nela que os argumentos necessários para a conexão são obtidos, via linha de comando. Portanto, a função `main` deve ficar da seguinte forma:

```
int main(int argc, char *argv[]) {
    char *nome;
    struct sockaddr_in enderecoServidor;
    struct hostent *host;
    long porta;

    if (argc != 4) {
        fprintf(stderr, "Considere executar o programa da seguinte
forma:\n./executavel [usuario] [host] [porta]\n");
        exit(1);
    }

    nome = argv[1];

    printf("Ativando o cliente...\n");

    // Obtém informações sobre o host (nome do servidor) a partir do DNS.
    if ((host = gethostbyname(argv[2])) == NULL) {
        fprintf(stderr, "Não foi possível obter o nome do host.\n");
        exit(1);
    }

    porta = atoi(argv[3]);

    // Cria um socket para a comunicação com o servidor.
    if ((socketFd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Não foi possível criar o socket.\n");
        exit(1);
    }
}
```

```

// Configura e conecta ao servidor.
configurarEConectar(&enderecoServidor, host, socketFd, porta);

printf("Cliente conectado!\n");

return 0;
}

```

Acrescentou-se uma verificação no nome do *host*, deixando assim o código menos suscetível a erros, caso um *host* não definido ou não encontrado seja informado. Além disso, o descritor de arquivo do *socket*, declarado globalmente (*socketFd*), finalmente cria um *socket* de fato através da função *socket*. Nesta função, informa-se que o protocolo será o IPv4 (*AF\_INET*) e que a conexão será sequencial e bidirecional, utilizando o TCP (*SOCK\_STREAM*). Depois, a função *configurarEConectar* utiliza os parâmetros informados para estabelecer a conexão.

É possível perceber que, ao tentar estabelecer uma conexão com o servidor completo, alguns problemas acontecem. Embora a conexão seja bem sucedida e o ingresso do usuário no *chat* seja registrado, imediatamente as duas aplicações são encerradas. Isso será corrigido na sequência.

### 3.4. Implementação da Função *loopChat*

Nesse passo o *chat* começa a funcionar! O código abaixo contém a implementação das funções *construirMensagem* e *loopChat*:

```

void construirMensagem(char *resultado, char *nome, char *msg) {
    char timestamp[12];
    time_t rawtime;
    struct tm *info;
    time(&rawtime);
    info = localtime(&rawtime);

    strftime(timestamp, sizeof(timestamp), "[%T] ", info); // Formata o timestamp

    memset(resultado, 0, TAMANHO_BUFFER);
    strcpy(resultado, timestamp);
    strcat(resultado, "@");
    strcat(resultado, nome);
    strcat(resultado, " diz: ");
    strcat(resultado, msg);
}

```



```

}

void loopChat(char *nome, int socketFd) {
    fd_set descritoresCliente;
    char mensagemChat[TAMANHO_BUFFER];
    char bufferChat[TAMANHO_BUFFER], bufferMensagem[TAMANHO_BUFFER];

    while (1) {
        FD_ZERO(&descritoresCliente);
        FD_SET(socketFd, &descritoresCliente);
        FD_SET(0, &descritoresCliente);

        if (select(FD_SETSIZE, &descritoresCliente, NULL, NULL, NULL) != -1) {
            for (int fd = 0; fd < FD_SETSIZE; fd++) {
                if (FD_ISSET(fd, &descritoresCliente)) {
                    if (fd == socketFd) {
                        int numBytesLidos = read(socketFd, bufferMensagem,
TAMANHO_BUFFER - 1);

                        bufferMensagem[numBytesLidos] = '\0';
                        printf("%s", bufferMensagem);
                        memset(&bufferMensagem, 0, sizeof(bufferMensagem));
                    }
                    else if (!fd) {
                        fgets(bufferChat, TAMANHO_BUFFER - 1, stdin);
                        construirMensagem(mensagemChat, nome, bufferChat);

                        if (write(socketFd, mensagemChat, TAMANHO_BUFFER - 1) ==
-1)

                            perror("Falha ao escrever no socket: ");
                        memset(&bufferChat, 0, sizeof(bufferChat));
                    }
                }
            }
        }
    }
}

```

O objetivo da função `construirMensagem` é apenas formatar a mensagem exibida pelos usuários. Definiu-se que a mensagem terá o formato `[HH:MM:SS] @nome-usuario diz: mensagem`. Essa função é chamada na função `loopChat`, que é a função do *loop* principal do *chat*.

A função `loopChat` monitora a entrada do usuário (`stdin`) e o `socket` para mensagens do servidor simultaneamente. Ela usa a função `select` para aguardar a disponibilidade de leitura em qualquer um desses descritores. Se houver dados para ler em algum deles, a função processa a entrada do usuário ou mensagens do servidor. Com a implementação desta função, basta inserir na função `main` a chamada `loopChat(nome, socketFd);` antes da instrução de retorno (`return 0;`).

O *loop* presente na função `loopChat` estará constantemente executando, e embora utilize a função `select` para monitorar os descritores de arquivo, ele continua verificando repetidamente se deve agir com base nas entradas. Portanto, enquanto o `select` ajuda a evitar a espera ocupada, o *loop* ainda representa uma forma de espera ativa, consumindo recursos da CPU.

Com o código apresentado é possível fazer a aplicação de *chat* funcionar, mas alguns problemas ainda são encontrados. Não há uma função específica para o usuário sair do *chat*, e isso atrapalha o funcionamento do servidor. Ao encerrar a aplicação do cliente com `CTRL+C`, o servidor imprime lixo e também é encerrado. Além disso, o descritor do `socket` não está sendo encerrado. Isso mantém a porta em uso e impede que o servidor seja executado novamente na mesma porta por algum tempo.

Outro ponto importante é configurar descritores de arquivo para operações não bloqueantes. Isso é útil em situações em que deseja-se que operações de leitura e escrita não bloqueiem a execução do programa enquanto aguardam a conclusão da operação, ou seja, quando não é desejável que a espera possa causar uma paralisação indesejada do programa. Em uma situação como a de uma aplicação de *chat*, deseja-se manter a capacidade de receber mensagens sem bloquear a execução, mesmo se não houver mensagens imediatamente disponíveis. Se a operação de leitura estiver configurada para bloquear, o programa pode ficar paralisado até que uma mensagem seja recebida, o que não é desejado em aplicativos de comunicação em tempo real. Portanto, um passo necessário é definir explicitamente que os descritores devem ser não bloqueantes.

### 3.5. Descritores Não Bloqueantes e Tratador de Interrupção

A função que define os descritores como não bloqueantes deve ser implementada da seguinte maneira:

```
void definirNaoBloqueante(int fd) {  
    int flags = fcntl(fd, F_GETFL);
```

```

    if (flags < 0)
        perror("fcntl falhou");

    flags |= O_NONBLOCK;
    fcntl(fd, F_SETFL, flags);
}

```

Assim, um descritor de arquivos passado como parâmetro será não bloqueante. Em geral, `stdin` (parâmetro 0) costuma ser não bloqueante, mas a função será chamada tanto para `socketFd` quanto para 0.

A função de tratamento de interrupção deve ser simples:

```

void manipuladorInterrupcao(int sigNaoUsado) {
    if (write(socketFd, "/sair\n", 6) == -1)
        perror("Falha ao escrever no socket: ");

    close(socketFd);
    exit(1);
}

```

No código, escreve-se `/sair` no `socket`, encerra-se o `socket` e a aplicação. Desta forma, resolve-se o problema de manter ativa a porta utilizada na conexão ora encerrada. Essa função precisa ser chamada em duas situações:

1. Quando ocorre explicitamente a digitação do comando de saída por parte do usuário: `/sair` ou `/sair [texto]` (**obs.:** `/sairtexto` qualquer não deve encerrar a aplicação); e
2. Quando ocorre uma interrupção gerada pelo sinal `SIGINT`, ou seja, quando o usuário pressiona `CTRL+C`.

Para resolver o caso 1, basta alterar o trecho de `loopChat` que constrói a mensagem, podendo receber a mensagem a ser construída ou o comando de saída:

```

if (!strncmp(bufferChat, "/sair\n", 6) || !strncmp(bufferChat, "/sair ", 6))
    manipuladorInterrupcao(-1);
else {
    construirMensagem(mensagemChat, nome, bufferChat);
    (...)
}

```

Para resolver o caso 2, é necessário apenas chamar a função `signal` informando o tipo de sinal e a função que o tratará.

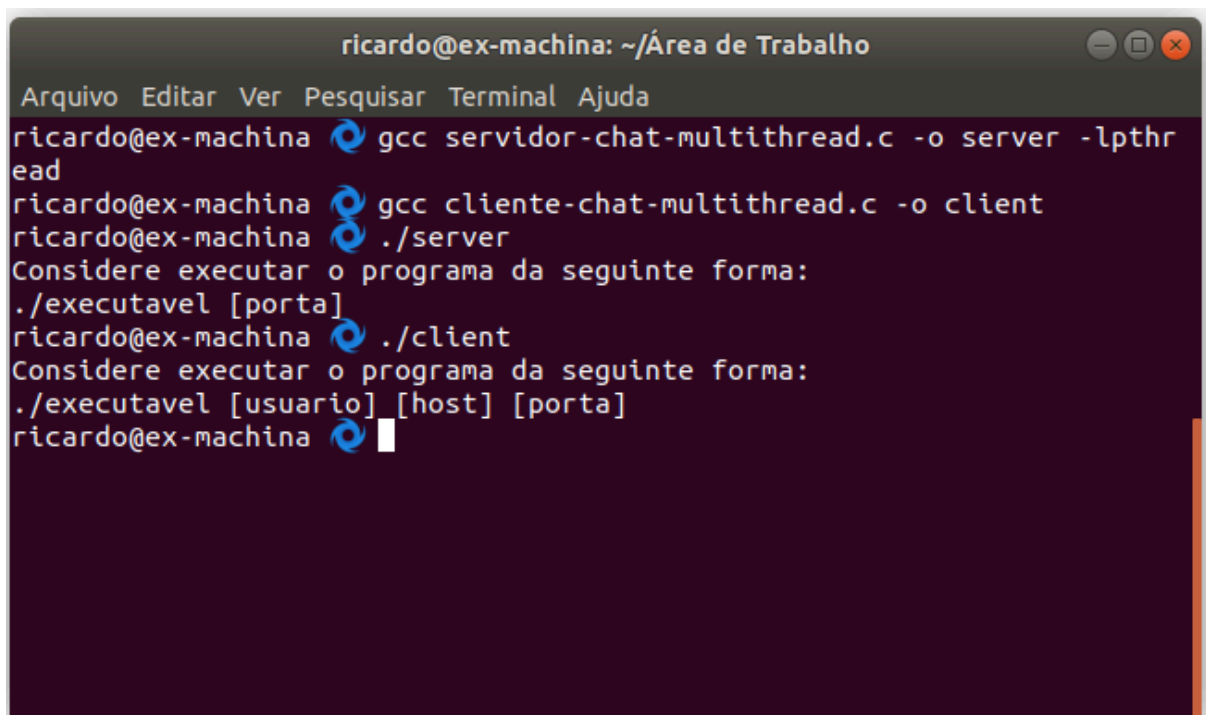
Na função `main`, basta acrescentar o seguinte código antes da chamada à função `loopChat`:

```
definirNaoBloqueante(socketFd);  
definirNaoBloqueante(0);  
signal(SIGINT, manipuladorInterrupcao);
```

#### 4. Teste da Aplicação

A aplicação foi testada tanto localmente quanto na rede do laboratório do IFC. Em ambos os testes utilizou-se o nome `server` para o arquivo executável do servidor e `client` para o cliente. Foram abertos três terminais para clientes.

A Figura 3 exibe o teste de compilação do servidor e do cliente e a tentativa de execução sem informar os parâmetros, caso em que os executáveis avisam corretamente o usuário para que proceda com a entrada desses dados.

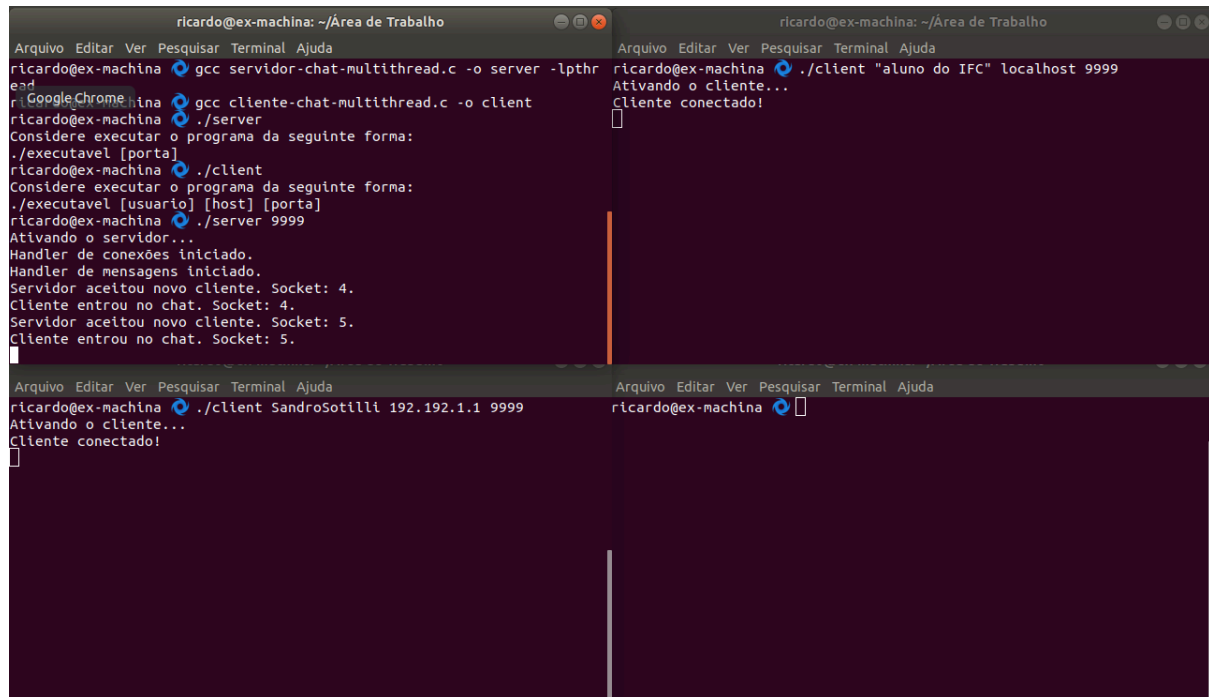


```
ricardo@ex-machina: ~/Área de Trabalho  
Arquivo Editar Ver Pesquisar Terminal Ajuda  
ricardo@ex-machina gcc servidor-chat-multithread.c -o server -lpthread  
ricardo@ex-machina gcc cliente-chat-multithread.c -o client  
ricardo@ex-machina ./server  
Considere executar o programa da seguinte forma:  
./executavel [porta]  
ricardo@ex-machina ./client  
Considere executar o programa da seguinte forma:  
./executavel [usuario] [host] [porta]  
ricardo@ex-machina
```

**Figura 3.** Compilação e teste dos arquivos `server` e `client`. Fonte: elaborado pelo autor.

Após esta ação, o teste seguinte consiste em executar corretamente o servidor e conectar dois clientes a ele. Em ambas aplicações é necessário informar os parâmetros informados

na Figura 3. Nota-se observando a Figura 4 que isso pode ser feito de algumas maneiras diferentes, tais como informando um nome de usuário entre aspas e informando o *host* de algumas formas diferentes. No exemplo da figura, conectou-se chamando o servidor de *localhost* e de 192.192.1.1 (endereço fictício), representando o endereço IP do servidor. É possível ainda utilizar o endereço *loopback*<sup>4</sup> 127.0.0.1, que faz exatamente o mesmo que *localhost*.



```
ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina gcc servidor-chat-multithread.c -o server -lpthr
ricardo@ex-machina gcc cliente-chat-multithread.c -o client
ricardo@ex-machina ./server
Considere executar o programa da seguinte forma:
./executavel [porta]
ricardo@ex-machina ./client
Considere executar o programa da seguinte forma:
./executavel [usuario] [host] [porta]
ricardo@ex-machina ./server 9999
Ativando o servidor...
Handler de conexões iniciado.
Handler de mensagens iniciado.
Servidor aceitou novo cliente. Socket: 4.
Cliente entrou no chat. Socket: 4.
Servidor aceitou novo cliente. Socket: 5.
Cliente entrou no chat. Socket: 5.

ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina ./client "aluno do IFC" localhost 9999
Ativando o cliente...
Cliente conectado!

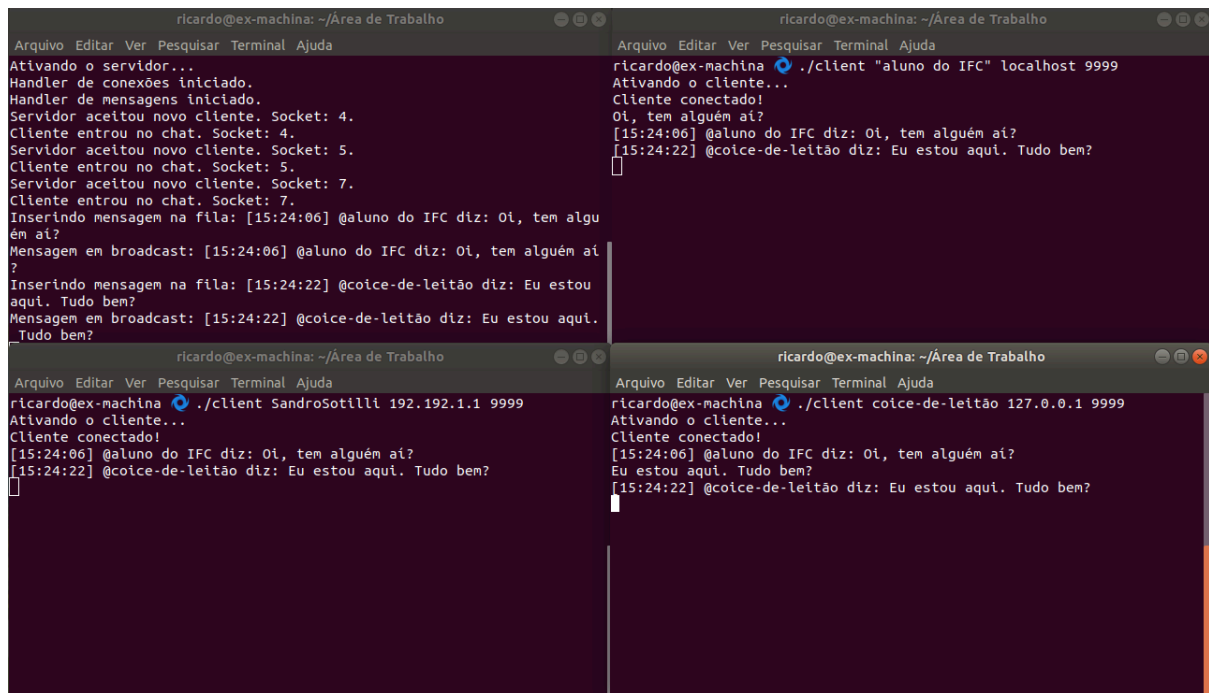
ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina ./client SandroSotilli 192.192.1.1 9999
Ativando o cliente...
Cliente conectado!
```

**Figura 4.** Execução correta dos arquivos *server* e *client*. Fonte: elaborado pelo autor.

A Figura 5 exibe o exemplo completo com três clientes interagindo pelo *chat* e as ações do servidor. É possível observar que o servidor exibe na tela o número do *socket* de cada cliente conectado e insere na fila de mensagens a mensagem de cada usuário, em ordem de chegada. As mensagens são enviadas em *broadcast* no formato definido na função *construirMensagem*, conforme visto no código do cliente.

---

<sup>4</sup>Endereço que faz a entrega de pacotes de volta para o próprio *host*.



The image displays four terminal windows arranged in a 2x2 grid, all with the title bar 'ricardo@ex-machina: ~/Área de Trabalho'. The top-left window shows the server's startup sequence: 'Ativando o servidor...', 'Handler de conexões iniciado.', 'Handler de mensagens iniciado.', and then it logs three clients connecting via sockets 4, 5, and 7. It then shows a broadcast message from '@aluno do IFC' and another from '@coice-de-leitão'. The top-right window shows a client running './client "aluno do IFC" localhost 9999', connecting, and sending two messages. The bottom-left window shows a client running './client SandroSotilli 192.192.1.1 9999', connecting, and sending two messages. The bottom-right window shows a client running './client coice-de-leitão 127.0.0.1 9999', connecting, and sending two messages.

```
ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
Ativando o servidor...
Handler de conexões iniciado.
Handler de mensagens iniciado.
Servidor aceitou novo cliente. Socket: 4.
Cliente entrou no chat. Socket: 4.
Servidor aceitou novo cliente. Socket: 5.
Cliente entrou no chat. Socket: 5.
Servidor aceitou novo cliente. Socket: 7.
Cliente entrou no chat. Socket: 7.
Inserindo mensagem na fila: [15:24:06] @aluno do IFC diz: Oi, tem algu
ém aí?
Mensagem em broadcast: [15:24:06] @aluno do IFC diz: Oi, tem alguém aí
?
Inserindo mensagem na fila: [15:24:22] @coice-de-leitão diz: Eu estou
aqui. Tudo bem?
Mensagem em broadcast: [15:24:22] @coice-de-leitão diz: Eu estou aqui.
Tudo bem?

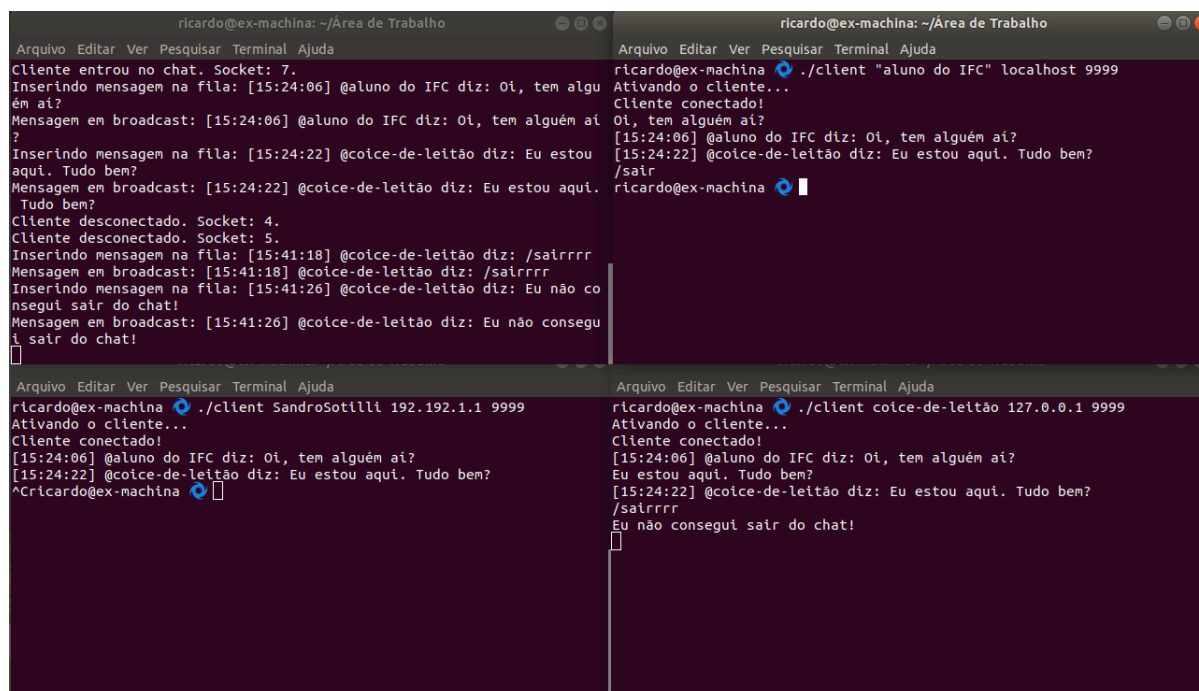
ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina ./client "aluno do IFC" localhost 9999
Ativando o cliente...
Cliente conectado!
[15:24:06] @aluno do IFC diz: Oi, tem alguém aí?
[15:24:22] @coice-de-leitão diz: Eu estou aqui. Tudo bem?

ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina ./client SandroSotilli 192.192.1.1 9999
Ativando o cliente...
Cliente conectado!
[15:24:06] @aluno do IFC diz: Oi, tem alguém aí?
[15:24:22] @coice-de-leitão diz: Eu estou aqui. Tudo bem?

ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina ./client coice-de-leitão 127.0.0.1 9999
Ativando o cliente...
Cliente conectado!
[15:24:06] @aluno do IFC diz: Oi, tem alguém aí?
Eu estou aqui. Tudo bem?
[15:24:22] @coice-de-leitão diz: Eu estou aqui. Tudo bem?
```

**Figura 5.** Interação entre usuários do *chat* e controle do servidor. Fonte: elaborado pelo autor.

Um usuário que deseja sair, como definido no escopo da aplicação, pode fazê-lo utilizando as teclas `CTRL+C` ou digitando `/sair [mensagem]`. A Figura 6 mostra os três usuários tentando sair do *chat*. O usuário cujo terminal está à direita do servidor sai corretamente digitando o comando de saída (`/sair`). O cliente cujo terminal está abaixo do servidor encerra a aplicação por meio da interrupção gerada pelo sinal `SIGINT`. O outro usuário tenta sair digitando `/sairrrrr`, o que não encerra a aplicação e o mantém conectado. O servidor exibe na tela o *socket* dos usuários desconectados e assim controla quantos clientes estão conectados a ele.



The image displays four terminal windows arranged in a 2x2 grid, all showing the output of a chat application. The top-left window shows a client connecting to localhost 9999, sending messages, and disconnecting. The top-right window shows a client connecting to localhost 9999, sending messages, and disconnecting. The bottom-left window shows a client connecting to 192.192.1.1 9999, sending messages, and disconnecting. The bottom-right window shows a client connecting to 127.0.0.1 9999, sending messages, and disconnecting.

```
ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
Cliente entrou no chat. Socket: 7.
Inserindo mensagem na fila: [15:24:06] @aluno do IFC diz: Oi, tem algu
ém aí?
Mensagem em broadcast: [15:24:06] @aluno do IFC diz: Oi, tem alguém aí
?
Inserindo mensagem na fila: [15:24:22] @coice-de-leitão diz: Eu estou
aqui. Tudo bem?
Mensagem em broadcast: [15:24:22] @coice-de-leitão diz: Eu estou aqui.
Tudo bem?
Cliente desconectado. Socket: 4.
Cliente desconectado. Socket: 5.
Inserindo mensagem na fila: [15:41:18] @coice-de-leitão diz: /sairrrr
Mensagem em broadcast: [15:41:18] @coice-de-leitão diz: /sairrrr
Inserindo mensagem na fila: [15:41:26] @coice-de-leitão diz: Eu não co
nsegui sair do chat!
Mensagem em broadcast: [15:41:26] @coice-de-leitão diz: Eu não consegu
i sair do chat!
[]

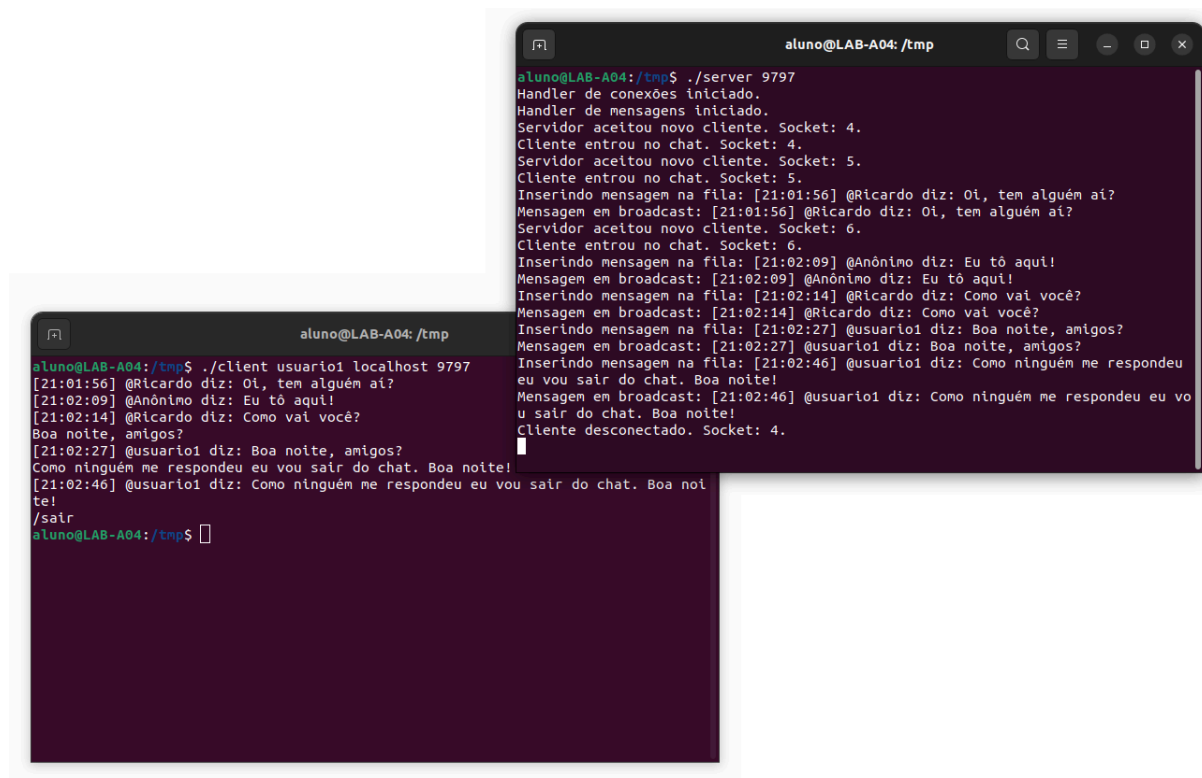
ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina: ./client "aluno do IFC" localhost 9999
Ativando o cliente...
Cliente conectado!
Oi, tem alguém aí?
[15:24:06] @aluno do IFC diz: Oi, tem alguém aí?
[15:24:22] @coice-de-leitão diz: Eu estou aqui. Tudo bem?
/sair
ricardo@ex-machina: []

ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina: ./client SandroSotilli 192.192.1.1 9999
Ativando o cliente...
Cliente conectado!
[15:24:06] @aluno do IFC diz: Oi, tem alguém aí?
[15:24:22] @coice-de-leitão diz: Eu estou aqui. Tudo bem?
^Cricardo@ex-machina: []

ricardo@ex-machina: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
ricardo@ex-machina: ./client coice-de-leitão 127.0.0.1 9999
Ativando o cliente...
Cliente conectado!
[15:24:06] @aluno do IFC diz: Oi, tem alguém aí?
Eu estou aqui. Tudo bem?
[15:24:22] @coice-de-leitão diz: Eu estou aqui. Tudo bem?
/sairrrrr
Eu não consegui sair do chat!
[]
```

**Figura 6.** Saída dos usuários do *chat*. Fonte: elaborado pelo autor.

Um servidor também foi ativado a partir de uma máquina do laboratório do IFC. Testou-se o acesso a esse servidor por meio do mesmo computador (`localhost`) e também de outros computadores da mesma rede. A Figura 7 traz uma captura de tela de dois terminais do computador que hospedou a aplicação servidor, um com a própria tela do servidor e outro com um cliente. Como pode-se ver na imagem, o servidor recebeu a conexão de três usuários, ou seja, o usuário local e outros dois usuários que o acessaram de outras máquinas. As mensagens trocadas pelos usuários foram corretamente entregues. Finalmente, a imagem mostra também o usuário local se desconectando através do comando `/sair` e o servidor processando essa informação.



The image shows two terminal windows. The top window, titled 'aluno@LAB-A04: /tmp', shows the execution of a server program. It starts with './server 9797' and displays various status messages like 'Handler de conexões iniciado.', 'Handler de mensagens iniciado.', and 'Servidor aceitou novo cliente. Socket: 4.'. It then shows a series of messages being received and broadcasted, including 'Ricardo diz: Oi, tem alguém aí?' and '@Anônimo diz: Eu tô aqui!'. The bottom window, also titled 'aluno@LAB-A04: /tmp', shows the execution of a client program. It starts with './client usuario1 localhost 9797' and displays messages received from the server, such as '@Ricardo diz: Oi, tem alguém aí?' and '@Anônimo diz: Eu tô aqui!'. The client also shows messages being sent back to the server, including 'Boa noite, amigos?' and 'Como ninguém me respondeu eu vou sair do chat. Boa noite!'.

**Figura 7.** Execução do servidor e acesso de usuários por meio da rede interna do IFC. Fonte: elaborado pelo autor.

Para reproduzir essa situação, é importante que a máquina que hospeda o servidor permita conexões na porta utilizada. Para isso, é importante verificar, entre outras coisas, se um *firewall* (tal como *iptables* ou *nftables*) está ativo.

Caso um *firewall* esteja ativo, possivelmente a porta não estará habilitada a receber conexões que não sejam locais. Para isso, pode ser necessário habilitá-las. Caso o usuário utilize *ufw*, é necessário digitar

```
> sudo ufw allow [porta]
```

Onde [porta] é a porta em que o servidor escutará. Depois disso, `sudo ufw status` mostrará as portas habilitadas. Caso o usuário deseje remover uma das regras, é necessário digitar `sudo ufw status numbered` para obter o ID de cada regra. Exemplo:

```
> sudo ufw status numbered
```

Estado: ativo

Para	Ação	De
----	----	--



```
[ 1] 443/tcp          ALLOW IN    Anywhere
[ 2] 9999             ALLOW IN    Anywhere
[ 3] 443/tcp (v6)     ALLOW IN    Anywhere (v6)
[ 4] 9999 (v6)       ALLOW IN    Anywhere (v6)
```

Assim, caso o usuário deseje remover novamente a possibilidade de liberar acesso pela porta 9999, é necessário remover as regras 4 e 2:

```
> sudo ufw delete 4
Apagando:
  allow 9999
Proceder com operação (s|n)? s
Regra apagada (v6)
> sudo ufw delete 2
Apagando:
  allow 9999
Proceder com operação (s|n)? s
Regra apagada
```

## 5. Conceitos Aplicados

A criação de uma aplicação como um programa de bate-papo permite aplicar diversos conceitos, não só de Programação de Alto Desempenho, mas também de outras áreas da Ciência da Computação.

Entre os conceitos de Programação de Alto Desempenho já trabalhados, é possível citar

- O estudo sobre **processos**, fundamental para compreender a natureza da comunicação e coordenação entre as diferentes entidades em um sistema operacional. Essa compreensão é refletida na estrutura da aplicação criada, onde o servidor é um processo dedicado à escuta e tratamento de múltiplas conexões, enquanto os clientes são processos individuais responsáveis pela interação com os usuários. A abordagem de processos permite a segregação eficaz das operações, possibilitando a execução concorrente das atividades do servidor, como aceitar novas conexões e gerenciar mensagens.
- o uso de **sockets**, permitindo a **comunicação entre processos** na rede. Em uma aplicação que segue a arquitetura cliente-servidor, tal como o chat proposto, os **sockets** permitem a comunicação entre diferentes **processos** (programas em execução) em diferentes máquinas, utilizando a rede como meio de comunicação.

- o uso de **threads**, lidando com várias conexões de clientes simultaneamente. Cada cliente tem sua própria *thread* dedicada para processar mensagens, o que evita que a comunicação com um cliente bloqueie as interações com outros clientes. Essa é uma abordagem eficiente em termos de uso de **recursos do sistema** em uma aplicação de *chat*, onde várias mensagens podem estar chegando ao servidor simultaneamente.
- o uso de **mutexes** e **variáveis condicionais**, recursos de **sincronização**, que garantiram a **exclusão mútua** nas **seções críticas** do código, que são o acesso à lista de usuários (clientes) ativos e à fila de mensagens.
- o **uso eficiente de recursos**, permitindo que o servidor lide simultaneamente com muitos clientes. Por exemplo: se uma *thread* estiver bloqueada aguardando uma operação de E/S, outras *threads* podem continuar seu processamento, mantendo a CPU ocupada.

Apesar da citação ao uso eficiente de recursos, relata-se que a estratégia de implementação com *espera ocupada*, adotada em alguns trechos dos códigos das aplicações servidor e cliente, se substituída por outra como *poll* ou alguma abordagem *orientada a eventos*, poderia proporcionar ainda mais eficiência no uso de recursos.

É possível citar ainda outros conceitos trabalhados no curso de Ciência da Computação, em outros componentes curriculares, relevantes para o desenvolvimento da aplicação vista, tais como

- *Programação Modular e Estruturada em Linguagem C* (**Algoritmos, Programação e Paradigmas de Programação**)
- *O Modelo Cliente-Servidor* (**Redes de Computadores**)
- *Protocolos de Rede* (**Redes de Computadores**)
- *Criação e Manipulação de Fila* (**Estruturas de Dados**)
- *Gerenciamento de Memória* (**Arquitetura de Computadores e Sistemas Operacionais**)
- *Tratamento de Interrupções* (**Arquitetura de Computadores e Sistemas Operacionais**)
- *Gerenciamento de Arquivos* (**Sistemas Operacionais**)
- *Comunicação e Sincronização de Processos e Exclusão Mútua* (também vistos em **Sistemas Distribuídos**)
- (...)

## Código do Servidor

Os links a seguir contêm os códigos passo a passo do servidor:

- [servidor-chat-multithread-passo-1.c](#)
- [servidor-chat-multithread-passo-2.c](#)
- [servidor-chat-multithread-passo-3.c](#)
- [servidor-chat-multithread-passo-4.c](#)
- [servidor-chat-multithread.c](#) (código completo também disponível na sequência)

```
// Servidor

// Execute com ./executavel [porta], onde
// [porta] deve ser um número acima de 1023, por exemplo, 1234.
// Ela será a porta na qual o servidor escutará e receberá conexões.

#include <arpa/inet.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#define TAMANHO_BUFFER 1024

// Struct que representa uma fila de mensagens usando um array de caracteres.
typedef struct {
    char *buffer[TAMANHO_BUFFER]; // Armazena as mensagens na fila.
    int inicio, cauda; // Índices para controlar a fila.
    int cheia, vazia; // Flags que indicam se a fila está cheia ou vazia.
    pthread_mutex_t *mutex; // Mutex para sincronização.
    pthread_cond_t *naoCheia, *naoVazia; // Variáveis de condição para
sincronização.
} Fila;

// Struct contendo dados importantes para o funcionamento do servidor.
typedef struct {
    fd_set descritoresLeituraServidor; // Conjunto de descritores de leitura para o
servidor.
    int socketFd; // Descritor do socket do servidor.
```

```

    int socketsClientes[TAMANHO_BUFFER]; // Array de descritores de sockets dos
    clientes.

    int numClientes; // Número atual de clientes conectados.
    pthread_mutex_t *mutexListaClientes; // Mutex para lista de clientes.
    Fila *filaMensagens; // Fila de mensagens do servidor.
} DadosServidor;

// Struct simples para armazenar DadosServidor e o novo socket do cliente.
typedef struct {
    DadosServidor *dados;
    int socketCliente;
} DadosHandlerCliente;

// Inicializa a fila de mensagens do servidor.
Fila* inicializarFila() {
    // Aloca memória para a estrutura de fila.
    Fila *fila = (Fila *)malloc(sizeof(Fila));
    if (fila == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }

    // Inicializa os valores da fila.
    fila->vazia = 1;
    fila->cheia = fila->inicio = fila->cauda = 0;

    // Aloca e inicializa um mutex para proteger operações na fila.
    fila->mutex = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
    if (fila->mutex == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }
    pthread_mutex_init(fila->mutex, NULL);

    // Aloca e inicializa variáveis condicionais para sinalização de fila não cheia
    e não vazia.
    fila->naoCheia = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
    if (fila->naoCheia == NULL) {
        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }
    pthread_cond_init(fila->naoCheia, NULL);

    fila->naoVazia = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
    if (fila->naoVazia == NULL) {

```

```

        perror("Não foi possível alocar mais memória!");
        exit(EXIT_FAILURE);
    }

    pthread_cond_init(&fila->naoVazia, NULL);

    return fila;
}

// Insere uma mensagem no final da fila de mensagens do servidor.
void enqueue(Fila *fila, char* mensagem) {
    // Armazena a mensagem no buffer da fila na posição da cauda.
    fila->buffer[fila->cauda] = mensagem;

    // Move a posição da cauda para o próximo índice no buffer circular.
    fila->cauda++;

    // Verifica se a cauda atingiu o final do buffer circular e, se sim, volta para
    o início.
    if (fila->cauda == TAMANHO_BUFFER)
        fila->cauda = 0;

    // Se a cauda alcançou a cabeça da fila, significa que a fila está cheia.
    if (fila->cauda == fila->inicio)
        fila->cheia = 1;

    // Marca a fila como não vazia.
    fila->vazia = 0;
}

// Remove e retorna a mensagem mais antiga da fila de mensagens do servidor.
char* dequeue(Fila *fila) {
    // Obtém a mensagem mais antiga da fila (na posição da cabeça).
    char* mensagem = fila->buffer[fila->inicio];

    // Move a cabeça da fila para a próxima posição.
    fila->inicio++;

    // Verifica se a cabeça atingiu o final da fila e a reinicia se necessário.
    if (fila->inicio == TAMANHO_BUFFER)
        fila->inicio = 0;

    // Verifica se a fila está vazia após a remoção.
    if (fila->inicio == fila->cauda)
        fila->vazia = 1;
}

```

```

// Marca a fila como não cheia, pois um elemento foi removido.
fila->cheia = 0;

// Retorna a mensagem removida.
return mensagem;
}

// Libera a memória associada à fila.
void destruirFila(Fila *fila) {
    // Destrói o mutex associado à fila.
    pthread_mutex_destroy(fila->mutex);

    // Destrói as variáveis de condição associadas à fila (para controle de cheio e
    // vazio).
    pthread_cond_destroy(fila->naoCheia);
    pthread_cond_destroy(fila->naoVazia);

    // Libera a memória alocada para o mutex.
    free(fila->mutex);

    // Libera a memória alocada para as variáveis de condição.
    free(fila->naoCheia);
    free(fila->naoVazia);

    // Libera a memória alocada para a própria estrutura de fila.
    free(fila);
}

// Remove o socket da lista de sockets de cliente ativos e fecha-o.
void removerCliente(DadosServidor *dados, int socketCliente) {
    // Obtém o mutex para acessar com segurança a lista de clientes ativos.
    pthread_mutex_lock(dados->mutexListaClientes);

    // Procura o socket do cliente na lista de clientes ativos.
    for (int i = 0; i < TAMANHO_BUFFER; i++) {
        if (dados->socketsClientes[i] == socketCliente) {
            dados->socketsClientes[i] = 0;
            close(socketCliente);
            dados->numClientes--;

            // Sai do loop depois de encontrar o socket.
            i = TAMANHO_BUFFER;
        }
    }
}

```

```

    // Libera o mutex para permitir outros acessos concorrentes à lista de clientes
    ativos.
    pthread_mutex_unlock(dados->mutexListaClientes);
}

// Handler do cliente que lida com mensagens recebidas do cliente.
void *handlerCliente(void *dhc) {
    // Converte o ponteiro genérico de dados de handler do cliente de volta para o
    tipo DadosHandlerCliente.
    DadosHandlerCliente *dadosHandler = (DadosHandlerCliente *)dhc;
    DadosServidor *dadosServidor = (DadosServidor *)dadosHandler->dados;

    // Obtém a fila de mensagens e o socket do cliente deste handler.
    Fila *fila = dadosServidor->filaMensagens;
    int socketCliente = dadosHandler->socketCliente;

    char bufferMensagem[TAMANHO_BUFFER];
    while (1) {
        // Lê mensagens recebidas do cliente e armazena no buffer.
        int numBytesLidos = read(socketCliente, bufferMensagem, TAMANHO_BUFFER -
1);

        bufferMensagem[numBytesLidos] = '\0';

        // Se o cliente enviou "/sair\n", remove-o da lista de clientes e fecha o
        socket dele.
        if (!strcmp(bufferMensagem, "/sair\n", 6) || !strcmp(bufferMensagem,
"/sair ", 6)) {
            fprintf(stderr, "Cliente desconectado. Socket: %d.\n", socketCliente);
            removerCliente(dadosServidor, socketCliente);
            return NULL; // Encerra a thread do cliente.
        }
        else {
            // Espera até que a fila não esteja cheia antes de inserir uma
            mensagem.

            while (fila->cheia) {
                pthread_cond_wait(fila->naoCheia, fila->mutex);
            }

            // Obtém lock na fila, insere a mensagem na fila, libera o lock e
            sinaliza a variável condicional.
            pthread_mutex_lock(fila->mutex);
            fprintf(stderr, "Inserindo mensagem na fila: %s", bufferMensagem);
            enfileirar(fila, bufferMensagem);
            pthread_mutex_unlock(fila->mutex);

```

```

        pthread_cond_signal(fila->naoVazia); // Notifica o handler de mensagens
        que há uma nova mensagem na fila.
    }
}

// Thread para lidar com novas conexões. Adiciona o fd do cliente à lista de fds do
cliente e gera uma nova thread handlerCliente para ele.
void *novoHandlerCliente(void *dados) {
    // Converte o ponteiro genérico de dados de volta para o tipo DadosServidor.
    DadosServidor *dadosServidor = (DadosServidor *)dados;

    while(1) {
        // Aceita uma nova conexão de cliente e obtém o socket do cliente.
        int socketCliente = accept(dadosServidor->socketFd, NULL, NULL);

        // Verifica se a conexão foi aceita com sucesso.
        if(socketCliente > 0) {
            fprintf(stderr, "Servidor aceitou novo cliente. Socket: %d.\n",
socketCliente);

            // Obtém lock na lista de clientes.
            pthread_mutex_lock(dadosServidor->mutexListaClientes);

            // Verifica se o número de clientes ativos ainda não atingiu o limite.
            if (dadosServidor->numClientes < TAMANHO_BUFFER) {
                // Adiciona o novo cliente à lista de sockets de clientes ativos.
                for (int i = 0; i < TAMANHO_BUFFER; i++) {
                    // Encontra uma posição vazia na lista de sockets de clientes.
                    if (!FD_ISSET(dadosServidor->socketsClientes[i],
&(dadosServidor->descritoresLeituraServidor))) {
                        dadosServidor->socketsClientes[i] = socketCliente;
                        i = TAMANHO_BUFFER; // Sai do loop.
                    }
                }

                // Adiciona o novo socket à lista de descritores de leitura do
servidor.
                FD_SET(socketCliente,
&(dadosServidor->descritoresLeituraServidor));

                // Gera uma nova thread para lidar com mensagens do cliente.
                DadosHandlerCliente dhc;
                dhc.socketCliente = socketCliente;
                dhc.dados = dadosServidor;
            }
        }
    }
}

```



```

        pthread_t threadCliente;
        if (!pthread_create(&threadCliente, NULL, (void *)&handlerCliente,
(void *)&dhc)) {
            // Incrementa o contador de clientes ativos e exibe uma
mensagem.

            dadosServidor->numClientes++;
            fprintf(stderr, "Cliente entrou no chat. Socket: %d.\n",
socketCliente);
        }
        else
            close(socketCliente); // Fecha o socket em caso de falha na
criação da thread.
    }

    // Libera o lock na lista de clientes.
    pthread_mutex_unlock(dadosServidor->mutexListaClientes);
}
}
}

// Handler de mensagens que espera até que a fila de mensagens não esteja vazia e
envia em broadcast para os clientes.
void *handlerMensagens(void *dados) {
    // Converte os dados genéricos para o tipo de estrutura DadosServidor.
    DadosServidor *dadosServidor = (DadosServidor *)dados;

    // Obtém a referência para a fila de mensagens do servidor.
    Fila *fila = dadosServidor->filaMensagens;

    // Obtém a referência para o array de sockets dos clientes.
    int *socketsClientes = dadosServidor->socketsClientes;

    while(1) {
        // Obtém um bloqueio no mutex da fila.
        pthread_mutex_lock(fila->mutex);

        // Aguarda até que a fila de mensagens não esteja vazia.
        while(fila->vazia) {
            pthread_cond_wait(fila->naoVazia, fila->mutex);
        }

        // Remove a mensagem mais antiga da fila.
        char* mensagem = desenfileirar(fila);

```

```

        // Libera o bloqueio do mutex da fila e sinaliza que não está mais cheia.
        pthread_mutex_unlock(fila->mutex);
        pthread_cond_signal(fila->naoCheia);

        // Mensagem em broadcast para todos os clientes conectados.
        fprintf(stderr, "Mensagem em broadcast: %s", mensagem);

        // Envia a mensagem para todos os clientes ativos.
        for(int i = 0; i < dadosServidor->numClientes; i++) {
            int socket = socketsClientes[i];
            if (socket != 0 && write(socket, mensagem, TAMANHO_BUFFER - 1) == -1)
                perror("Falha na escrita do socket: ");
        }
    }
}

// Inicia as threads do servidor para lidar com conexões e mensagens.
void iniciarChat(int socketFd) {
    // Cria uma estrutura para armazenar os dados do servidor.
    DadosServidor dados;
    memset(dados.socketsClientes, 0, sizeof dados.socketsClientes);

    // Inicializa o número de clientes como zero.
    dados.numClientes = 0;

    // Configura o descritor de socket do servidor.
    dados.socketFd = socketFd;

    // Inicializa a fila de mensagens do servidor.
    dados.filaMensagens = inicializarFila();

    // Aloca memória e inicializa o mutex para a lista de clientes.
    dados.mutexListaClientes = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(dados.mutexListaClientes, NULL);

    // Inicia a thread para lidar com novas conexões de clientes.
    pthread_t threadConexao;
    if (!pthread_create(&threadConexao, NULL, (void *)&novoHandlerCliente, (void *)&dados)) {
        fprintf(stderr, "Handler de conexões iniciado.\n");
    }

    // Configura o conjunto de descritores de leitura do servidor.
    FD_ZERO(&(dados.descritoresLeituraServidor));
    FD_SET(socketFd, &(dados.descritoresLeituraServidor));

```

```

    // Inicia a thread para lidar com mensagens recebidas.
    pthread_t threadMensagens;
    if (!pthread_create(&threadMensagens, NULL, (void *)&handlerMensagens, (void
*) &dados)) {
        fprintf(stderr, "Handler de mensagens iniciado.\n");
    }

    // Aguarda a conclusão das threads de conexão e mensagens.
    pthread_join(threadConexao, NULL);
    pthread_join(threadMensagens, NULL);

    // Libera a memória utilizada pela fila de mensagens.
    destruirFila(dados.filaMensagens);

    // Destroi o mutex da lista de clientes e libera a memória associada.
    pthread_mutex_destroy(&dados.mutexListaClientes);
    free(dados.mutexListaClientes);
}

// Configura e associa o socket a um endereço IP e porta específicos.
void associarSocket(struct sockaddr_in *serverAddr, int socketFd, long port) {
    // Limpa a estrutura serverAddr para garantir que não haja dados residuais.
    memset(serverAddr, 0, sizeof(*serverAddr));

    // Define o tipo de endereço como IPv4.
    serverAddr->sin_family = AF_INET;

    // Configura o endereço IP para INADDR_ANY, que significa "escutar em todas as
interfaces de rede".
    serverAddr->sin_addr.s_addr = htonl(INADDR_ANY);

    // Configura a porta do servidor para a porta especificada.
    serverAddr->sin_port = htons(port);

    // Tenta associar o socket ao endereço e porta especificados.
    if (bind(socketFd, (struct sockaddr *)serverAddr, sizeof(struct sockaddr_in))
== -1) {
        perror("Falha ao associar o socket: ");
        exit(1);
    }
}

int main(int argc, char *argv[]) {
    // Declaração de uma estrutura para armazenar informações do servidor.

```

```

struct sockaddr_in serverAddr;

int porta;

// Declaração do descritor de arquivo do socket.
int socketFd;

// Verifica se foi fornecida uma porta na linha de comando.
if (argc == 2)
    porta = atoi(argv[1]);
else {
    fprintf(stderr, "Considere executar o programa da seguinte
forma:\n./executavel [porta]\n");
    exit(1);
}

printf("Ativando o servidor...\n");

// Cria um novo socket usando o protocolo IPv4 e TCP.
if ((socketFd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    perror("Falha na criação do socket.");
    exit(1);
}

// Associa o socket ao endereço e porta especificados.
associarSocket(&serverAddr, socketFd, porta);

// Inicia o servidor para ouvir novas conexões na porta especificada.
if (listen(socketFd, 1) == -1) {
    perror("Falha na escuta: ");
    exit(1);
}

// Inicia o chat, que lida com conexões e mensagens.
iniciarChat(socketFd);

// Fecha o socket após a execução.
close(socketFd);

return 0;
}

```

## Código do Cliente

Os *links* a seguir contêm os códigos passo a passo do cliente:

- [cliente-chat-multithread-passo-1.c](#)
- [cliente-chat-multithread-passo-2.c](#)
- [cliente-chat-multithread-passo-3.c](#)
- [cliente-chat-multithread-passo-4.c](#)
- [cliente-chat-multithread.c](#) (código completo também disponível na sequência)

```
// Cliente

// Execute com ./executavel [usuario] [host] [porta], onde
// [usuario] deve ser o seu nome de usuário.
// [host] deve ser o servidor.
// [porta] deve ser a porta na qual o servidor está escutando.

#include <arpa/inet.h>
#include <fcntl.h>
#include <netdb.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <time.h>
#include <unistd.h>

#define TAMANHO_BUFFER 1024

// Descritor de arquivo (file descriptor) do socket do servidor.
// O descritor de arquivo é um identificador numérico usado para acessar o socket
em operações de leitura, escrita e gerenciamento.
static int socketFd;

// Manipulador de interrupção para informar o servidor sobre o fechamento do socket
e a saída do programa
void manipuladorInterrupcao(int sigNaoUsado) {
    // Alterado para evitar o warning -Wstringop-overread (tentativa de escrita,
    neste caso, em posição de memória além do limite da string "\sair\n").
    // if (write(socketFd, "\sair\n", TAMANHO_BUFFER - 1) == -1)
    if (write(socketFd, "\sair\n", 6) == -1)
        perror("Falha ao escrever no socket: ");
}
```

```

    close(socketFd);
    exit(1);
}

// Constrói uma mensagem formatada com o nome do usuário e timestamp
void construirMensagem(char *resultado, char *nome, char *msg) {
    char timestamp[12]; // Tamanho suficiente para [HH:MM:SS] + espaço
    time_t rawtime;
    struct tm *info;
    time(&rawtime);
    info = localtime(&rawtime);

    strftime(timestamp, sizeof(timestamp), "[%T] ", info); // Formata o timestamp

    memset(resultado, 0, TAMANHO_BUFFER);
    strcpy(resultado, timestamp);
    strcat(resultado, "@");
    strcat(resultado, nome);
    strcat(resultado, " diz: ");
    strcat(resultado, msg);
}

// Loop principal do chat
void loopChat(char *nome, int socketFd) {
    fd_set descritoresCliente;
    char mensagemChat[TAMANHO_BUFFER];
    char bufferChat[TAMANHO_BUFFER], bufferMensagem[TAMANHO_BUFFER];

    while (1) {
        FD_ZERO(&descritoresCliente);
        FD_SET(socketFd, &descritoresCliente);
        FD_SET(0, &descritoresCliente);

        // Monitora eventos de leitura em descritores de arquivo (socket e stdin).
        if (select(FD_SETSIZE, &descritoresCliente, NULL, NULL, NULL) != -1) {
            for (int fd = 0; fd < FD_SETSIZE; fd++) {
                if (FD_ISSET(fd, &descritoresCliente)) {
                    if (fd == socketFd) {
                        int numBytesLidos = read(socketFd, bufferMensagem,
TAMANHO_BUFFER - 1);

                        bufferMensagem[numBytesLidos] = '\0';
                        printf("%s", bufferMensagem);
                        memset(&bufferMensagem, 0, sizeof(bufferMensagem));
                    }
                    else if (!fd) {

```

```

        fgets(bufferChat, TAMANHO_BUFFER - 1, stdin);

        // Verifica se o usuário digitou "/sair" ou "/sair [alguma
        coisa]" para sair do chat.
        if (!strncmp(bufferChat, "/sair\n", 6) ||
        !strncmp(bufferChat, "/sair ", 6))
            manipuladorInterrupcao(-1);
        else {
            construirMensagem(mensagemChat, nome, bufferChat);

            // Envia a mensagem para o servidor através do socket.
            if (write(socketFd, mensagemChat, TAMANHO_BUFFER - 1)
            == -1)

                perror("Falha ao escrever no socket: ");
                memset(&bufferChat, 0, sizeof(bufferChat));
            }
        }
    }
}

// Configura e conecta ao servidor
void configurarEConectar(struct sockaddr_in *enderecoServidor, struct hostent
*host, int socketFd, long porta) {
    memset(enderecoServidor, 0, sizeof(enderecoServidor));
    enderecoServidor->sin_family = AF_INET;
    enderecoServidor->sin_addr = *((struct in_addr *)host->h_addr_list[0]);
    enderecoServidor->sin_port = htons(porta);

    // Tenta estabelecer uma conexão com o servidor.
    if (connect(socketFd, (struct sockaddr *) enderecoServidor, sizeof(struct
sockaddr)) < 0) {
        perror("Não foi possível conectar ao servidor.");
        exit(1);
    }
}

// Define um descritor de arquivo como não bloqueante
void definirNaoBloqueante(int fd) {
    int flags = fcntl(fd, F_GETFL);
    if (flags < 0)
        perror("fcntl falhou");
}

```

```

    flags |= O_NONBLOCK;
    fcntl(fd, F_SETFL, flags);
}

void verificarNaoBloqueante(int fd) {
    int flags = fcntl(fd, F_GETFL);
    if (flags < 0) {
        perror("fcntl falhou");
    }
    else {
        if (flags & O_NONBLOCK)
            printf("O descritor de arquivo %d é não bloqueante.\n", fd);
        else
            printf("O descritor de arquivo %d não é não bloqueante.\n", fd);
    }
}

int main(int argc, char *argv[]) {
    char *nome;
    struct sockaddr_in enderecoServidor;
    struct hostent *host;
    long porta;

    if (argc != 4) {
        fprintf(stderr, "Considere executar o programa da seguinte
forma:\n./executavel [usuario] [host] [porta]\n");
        exit(1);
    }

    nome = argv[1];

    printf("Ativando o cliente...\n");

    // Obtém informações sobre o host (nome do servidor) a partir do DNS.
    if ((host = gethostbyname(argv[2])) == NULL) {
        fprintf(stderr, "Não foi possível obter o nome do host.\n");
        exit(1);
    }

    porta = atoi(argv[3]);

    // Cria um socket para a comunicação com o servidor.
    if ((socketFd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Não foi possível criar o socket.\n");
        exit(1);
    }
}

```



```

}

// Configura e conecta ao servidor.
configurarEConectar(&enderecoServidor, host, socketFd, porta);

printf("Cliente conectado!\n");

// Caso deseje ver se o padrão do fd é não bloqueante, você pode descomentar as
linhas abaixo.
// Assumindo que 0 seja não bloqueante, você pode remover a instrução
definirNaoBloqueante(0);
// verificarNaoBloqueante(socketFd);
// verificarNaoBloqueante(0);

// Define o socket e stdin como não bloqueantes.
definirNaoBloqueante(socketFd);
definirNaoBloqueante(0);

// Configura um manipulador de sinal para capturar interrupções e sair
controladamente.
signal(SIGINT, manipuladorInterrupcao);

// Inicia o loop principal do chat.
loopChat(nome, socketFd);

return 0;
}

```

## Resumo

Esta aula abordou a criação de uma ferramenta de bate-papo *multithread* em C para promover a aplicação prática dos conceitos vistos nas aulas anteriores. A aplicação utiliza processos e *sockets* para permitir a comunicação entre o servidor e os clientes, *threads* para lidar com várias conexões de clientes simultaneamente e sincronização por meio de *mutexes* e variáveis condicionais para garantir que as operações sejam seguras e coordenadas entre as *threads*. Esses conceitos são fundamentais para a construção de aplicativos de rede concorrentes e multiusuários.

## Exercícios

- 1) Execute o servidor e solicite que os colegas executem o cliente. Realizem conexões de clientes com o servidor e interajam utilizando a aplicação.
- 2) Implemente funcionalidades ainda não presentes na aplicação, tais como as que estão descritas na seção [Projeto e Escopo da Aplicação](#).
- 3) **(DESAFIO!)** Substitua a espera ocupada em todo código por outra abordagem mais eficiente.

## Referências

[1] LI, Eugene. Multithreaded-Chat-in-C. Disponível em:

<https://github.com/eugeneli/Multithreaded-Chat-in-C/tree/master>. Acesso em: 11 set. 2023.

[2] CASSANY, Daniel. Forums and chats to learn languages: functions, types and uses, in Rosario en Rosario Hernández y Paul Rankin (ed.), Higher Education and Second Language Learning. Promoting Self-Directed Learning in New Technological and Educational Contexts. Bern: Peter Lang.; p. 135-158. ISBN: 978-3-0343-1734-4. 2015. Disponível em:

[https://repositori.upf.edu/bitstream/handle/10230/56847/Cassany\\_hig\\_foru.pdf?sequence=1&isAllowed=y](https://repositori.upf.edu/bitstream/handle/10230/56847/Cassany_hig_foru.pdf?sequence=1&isAllowed=y). Acesso em: 07 ago. 2024.

[3] CRESSLER, Cosette. Understanding The Architecture & System Design Of A Chat Application. Disponível em:

<https://www.cometchat.com/blog/chat-application-architecture-and-system-design>. Acesso em: 12 set. 2023.