Sistema de múltiplas conexões

Execução: Individual
Data de entrega: 22 de junho de 2023 até 23h:59min

INTRODUÇÃO

Os chats inteligentes são soluções comerciais que já a tempos vêm sendo utilizadas para otimizar a experiência do cliente e desonerar atendentes quando a tarefa se trata de resolver uma simples dúvida, por exemplo. Com isso, essas empresas buscam reduzir custos significativos com recursos humanos, e até incentivar a pesquisa científica entre seus desenvolvedores. As evoluções desse tipo de solução não param, e a eficiência do serviço vem se tornando cada vez mais expressiva. A sensação do momento é o Chat GPT, tecnologia desenvolvida pela Microsoft que promete conseguir interpretar uma larga variedade de inputs e devolver respostas assertivas e igualmente abrangentes.

O gestor de uma startup chamada XPTO Inc. planeja o desenvolvimento de um chat inteligente, com o intuito de melhorar o serviço ao cliente, além de desafiar seus desenvolvedores com a construção dessa solução. Entretanto, antes de introduzir a inteligência, é necessário um primeiro contato com o conceito de "chat". Com isso, o gestor teve a ideia de desenvolver um sistema de chat em grupo utilizando uma rede local, conectando simultaneamente diversos clientes a um servidor central, como ilustra a Figura 1.

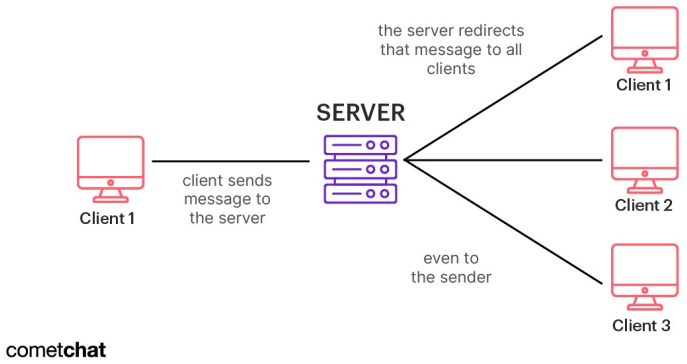

Figura 1 - Exemplo de arquitetura de aplicação de Chat disponibilizada pela cometchat.com

Neste trabalho prático, você será responsável por desenvolver um sistema que coordene múltiplas conexões simultâneas entre os clientes e permita a comunicação entre eles. Neste sistema, você deve desenvolver um **servidor**, responsável por coordenar as múltiplas conexões à medida em que os usuários entram e saem da rede, e **os clientes**, que trocam mensagens uns com os outros por intermédio do servidor. Toda conexão deve utilizar a interface de *sockets* na linguagem C.

Você desenvolverá 2 programas para um sistema simples de troca de mensagens de texto utilizando apenas as funcionalidades da biblioteca de *sockets* POSIX e a comunicação via protocolo TCP. O código do servidor deverá utilizar múltiplas *threads* para a manutenção das múltiplas conexões. As próximas seções detalham o que cada entidade (servidor ou usuário) deve fazer.

Os objetivos gerais deste trabalho são:

1. Implementar servidor utilizando a interface de *sockets* na linguagem C;
2. Implementar o cliente utilizando a interface de *sockets* na linguagem C;
3. Escrever o relatório;

PROTOCOLO

O protocolo de aplicação deverá funcionar sobre TCP. Isso implica que as mensagens serão entregues sobre um canal de bytes com garantias de entrega em ordem, mas é sua responsabilidade determinar onde começa e termina cada mensagem.

Especificação das Mensagens

Esta seção especifica as mensagens utilizadas na comunicação de controle e dados da rede, bem como as mensagens de erro e confirmação. Nas tabelas abaixo, as células em "–" correspondem aos campos que não precisam ser definidos nas mensagens. **As colunas em cinza não precisam ser implementadas.** Utilize o campo IdMsg para facilitar a identificação de cada tipo de requisição.

Comandos de Controle					
Type	**IdMsg**	**IdSender**	**IdReceiver**	**Message**	**Description**
REQ_ADD	01	–	–	–	Mensagem de requisição de inclusão de usuário na rede.
REQ_REM	02	IdUser	–	–	Mensagem de requisição de saída de um usuário da rede, onde IdUser, corresponde a identificação do usuário solicitante.
RES_LIST	04	–	–	$IdUser_j$, $IdUser_k$, …	Mensagem com lista de identificação de usuários, onde $IdUser_j$, $IdUser_k$, … correspondem aos identificadores dos usuários transmitidos na mensagem.

Comandos de Comunicação					
Type	**IdMsg**	**IdSender**	**IdReceiver**	**Message**	**Description**
MSG	06	$IdUser_i$	$IdUser_j$	Dados	Mensagem transmitida em broadcast para todos os integrantes da rede. O campo IdSender é obrigatório, sendo utilizado nas ocasiões de adição de usuário ao grupo e envio de mensagens para outros usuários. O campo IdReceiver é opcional, sendo utilizado somente na ocasião de uma mensagem privada ser enviada.
ERROR	07	–	$IdUser_j$	Code 01: "*User limit exceeded*" 02: "*User not found*" 03: "*Receiver not found*"	Mensagem de erro transmitida do Servidor para o usuário $IdUser_j$.
OK	08	–	$IdUser_j$	Code 01: "Removed Successfully"	Mensagem de confirmação transmitida do Servidor para o usuário $IdUser_j$.

Fluxo das Mensagens de Controle

Esta seção descreve o fluxo de mensagens de controle transmitidas entre os usuários e o servidor, a fim de coordenar a comunicação na rede, além das impressões em tela realizadas pelos usuários e o servidor.
* Os {Campos} entre chaves nas strings indicam uma variável e devem ser substituídos dinamicamente pelo parâmetro informado.
* As expressões sublinhadas são **exclusivas** entre aquelas de mesmo nível, e devem ser executadas quando a sua condição for atendida.

Abertura de conexão com Servidor

- Um Usuário **User$_i$** solicita para o Servidor a sua inclusão na rede através do comando **REQ_ADD.**
    - Se o número de usuários na rede já chegou ao seu limite, o Servidor responde **ERROR(01)**
        - Cliente imprime na tela a descrição do erro de código 01.
    - Se não, o Servidor define um identificador **IdUser$_i$** para o **User$_i$**, registra **IdUser$_i$** em sua base, imprime na tela a mensagem de confirmação “User {IdUser$_i$} added” e envia para todos os equipamentos conectados na rede (broadcast) a identificação do novo equipamento por meio da mensagem **MSG(IdUser$_i$, NULL, Message)**. Nesse caso, a mensagem de confirmação “User {IdUser$_i$} joined the group!” deve ser alocada ao parâmetro Message do comando.
        - Os usuários da rede recebem **MSG(IdUser$_i$, NULL, Message)**, registram **IdUser$_i$** em sua base de dados e imprimem na tela a mensagem “**{Message}**”.
        - Servidor envia a lista dos atuais integrantes do grupo para **User$_i$** (unicast) por meio da mensagem **RES_LIST(IdUser$_j$, IdUser$_j$, …)**.
            - Usuário **User$_i$** recebe **RES_LIST(IdUser$_j$, IdUser$_j$, …)** e registra novos usuários em sua base de dados.

Fechamento de Conexão com Servidor

- Um usuário **User$_i$** solicita para o Servidor o fechamento da conexão por meio da mensagem **REQ_REM(IdUser$_i$)**, que verifica por sua vez se **IdUser$_i$** existe na base de dados.
    - Se a resposta for negativa, o servidor responde mensagem de erro **ERROR(02)** para **User$_i$**.
        - O usuário imprime na tela a descrição do erro de código 02.
    - Se a resposta for positiva,
        - O Servidor remove **User$_i$** da base de dados, responde mensagem **OK(01)** para **User$_i$** , desconecta **User$_i$,** imprime em tela a mensagem “User {IdUser$_i$} removed”.
            - O usuário **User$_i$** recebe a mensagem **OK(01)**, imprime na tela o conteúdo da mensagem ok de código 01, fecha a conexão e encerra a execução.
        - O Servidor envia mensagem **REQ_REM(IdUser$_i$)** para os usuários que restaram no grupo (*broadcast*).
            - Os usuários restantes recebem **REQ_REM(IdUser$_i$)**, removem **IdUser$_i$** de suas bases de dados e imprimem em tela “User {IdUser$_i$} left the group!”.

Fluxo das Mensagens

Esta seção descreve o fluxo de mensagens trocadas entre os usuários do grupo e o servidor, além das decisões e impressões em tela.
* Os {Campos} entre chaves nas strings indicam uma variável e devem ser substituídos dinamicamente pelo parâmetro informado.
* As expressões sublinhadas são **exclusivas** entre aquelas de mesmo nível, e devem ser executadas quando a sua condição for atendida.

Mensagem Privada

- Um usuário **User$_i$** envia para o Servidor a mensagem que deseja que um outro usuário **User$_j$** leia por meio do comando **MSG(IdUser$_i$, IdUser$_j$)**. O servidor verifica se o **IdUser$_j$** existe em sua base de dados.
    - Caso não exista, o Servidor imprime na tela "User {IdUser$_j$} not found" e responde **ERROR(03)** para o usuário **User$_i$**.
        - O usuário **User$_i$** recebe **ERROR(03)** e imprime a descrição do erro de código 03.
    - Caso exista,
        - O usuário **User$_i$** printa a própria mensagem na tela devidamente identificada pelo Id do remetente **IdUser$_i$** e o horário de processamento da mensagem, além de uma indicação "P" de que a mensagem é privada e uma pequena setinha informando para qual ID a mensagem está sendo enviada: "P [hh:mm] -> {IdUser$_j$}: {Message}", onde hh são as horas e mm os minutos atuais do sistema.
        - O Servidor repassa **MSG(IdUser$_i$, IdUser$_j$, Message)** para **User$_j$**
            - O usuário **User$_j$** recebe a mensagem **MSG(IdUser$_i$, IdUser$_j$, Message)** e imprime o corpo da mensagem: "P [hh:mm] {IdUser$_i$}: {Message}".

Mensagem Pública

- Um usuário **User$_i$** envia para o Servidor a mensagem que deseja que a rede inteira veja usando o comando **MSG(IdUser$_i$, NULL, Message)**
    - O servidor printa a mensagem na tela e repassa via broadcast para todos os usuários do grupo
        - Os usuários da rede recebem **MSG(IdUser$_i$, NULL, Message)** e imprimem na tela a mensagem devidamente identificada pelo Id do remetente **IdUser$_i$**, além do horário de processamento da mensagem: "[hh:mm] {IdUser$_i$}: {Message}". O remetente da mensagem recebe uma versão alterada, com os caracteres -> all indicando que a mensagem foi enviada para todos: "[hh:mm] -> all {Message}"

IMPLEMENTAÇÃO

Pequenos detalhes devem ser observados no desenvolvimento de cada programa que fará parte do sistema. É importante observar que o protocolo é simples e único (o cliente sempre tem que enviar a mensagem codificada para o servidor e vice-versa, de modo que o correto entendimento da mensagem deve ser feito por todos os programas).

Como mencionado anteriormente, a implementação do protocolo da aplicação utilizará TCP. Haverá apenas um socket em cada equipamento (cliente), independente de quantos outros programas se comunicarem com aquele processo. O programador deve usar as funções *send* e *recv* para enviar e receber

mensagens. No caso do servidor, ele deve manter um socket para receber novas conexões (sobre o qual ele executará *accept*) e um socket para cada cliente conectado.

Tanto o IPv4 quanto o IPv6 serão cobrados neste trabalho prático. O **servidor** deve tratar até 15 conexões simultâneas. Ademais, o servidor é responsável por definir identificações únicas para cada usuário na rede. Um **usuário (cliente)** inicia sem identificação, e só a recebe após a solicitação de entrada na rede. Também, o equipamento deve receber mensagens do teclado. Tanto servidor quanto equipamento devem imprimir as mensagens recebidas na tela.

Comandos

O usuário deve receber alguns comandos pela entrada padrão (teclado) a fim de interagir com o sistema. Tais comando são descritos a seguir:

- **close connection:** Comando requisita fechamento de conexão, o que dispara o fluxo de mensagem de controle "Fechamento de Conexão com Servidor".
- **list users:** Comando lista os usuários na base da dados do usuário por id. (e.g. "**IdUser$_i$ IdUser$_j$ IdUser$_k$ ….**").
- **send to IdUser$_j$ {Message}:** Comando que manda uma mensagem privada **Message** para o usuário **IdUser$_j$**, o que dispara o fluxo de mensagem de dados "User$_i$ manda mensagem para User$_j$ na rede".
- **send all {Message}:** Comando que manda uma mensagem **Message** para todos os usuários em broadcast, o que dispara o fluxo de mensagem de dados "User$_i$ manda mensagem para o grupo na rede".

Execução

Seu servidor deve receber, **estritamente nessa ordem**, a versão do endereço entre "v4" ou "v6" e um número de porta na linha de comando especificando em qual porta ele vai receber conexões (Sugestão: utilize a porta 51511 para efeitos de padronização do trabalho). Seu equipamento (cliente) deve receber, **estritamente nessa ordem**, o endereço IP e a porta do servidor para estabelecimento da conexão. Para realizar múltiplas conexões de equipamentos com o servidor basta executar múltiplas vezes o código do equipamento.

A seguir, um exemplo de execução de dois equipamentos conectados com um servidor em três terminais distintos:

IPv4

```
Terminal 1: ./server v4 51511
Terminal 2: ./user 127.0.0.1 51511
Terminal 3: ./user 127.0.0.1 51511
```

IPv6

```
Terminal 1: ./server v6 51511
Terminal 2: ./user ::1 51511
Terminal 3: ./user ::1 51511
```

AVALIAÇÃO

O trabalho deve ser realizado individualmente e **deve ser implementado na linguagem de programação C** utilizando somente a biblioteca padrão (interface POSIX de sockets de redes). Deve ser possível executar

seu programa no sistema operacional **Linux** e **não deve utilizar bibliotecas Windows, como o winsock**. Seu programa deve interoperar com qualquer outro programa implementando o mesmo protocolo (você pode testar com as implementações dos seus colegas). Procure escrever seu código de maneira clara, com comentários pontuais e bem indentados. Isto facilita a correção dos monitores e tem impacto positivo na avaliação.

Entrega

Cada aluno deve entregar documentação em PDF de até 6 páginas, sem capa, utilizando fonte tamanho 10, e figuras de tamanho adequado ao tamanho da fonte. **Ele deve conter uma descrição da arquitetura adotada para o servidor, os refinamentos das ações identificadas no mesmo, as estruturas de dados utilizadas, as decisões de implementação não documentadas nesta especificação**. Como sugestão, considere incluir as seguintes seções no relatório: introdução, arquitetura, servidor, equipamento, discussão e conclusão. O relatório deve ser entregue em formato PDF. A documentação corresponde a 20% dos pontos do trabalho, mas só será considerada para as funcionalidades implementadas corretamente.

Será utilizado um sistema para detecção de código repetido, portanto não é admitido cola de trabalhos.
Será adotada a média harmônica entre as notas da documentação e da execução, o que implica que a nota final será 0 se uma das partes não for entregue.

Cada aluno deve entregar, além da documentação, o **código fonte em C** e um **Makefile** para compilação do programa. Instruções para submissão e compatibilidade com o sistema de correção semi-automática:

- O Makefile deve compilar o “user” e o “server”.
- Seu código deve ser compilado pelo comando “make” sem a necessidade de parâmetros adicionais.
- A entrega deve ser feita no formato ZIP, seguindo a nomenclatura: TP2_MATRICULA.zip
- O nome dos arquivos deve ser padronizado:
    - server.c
    - user.c
    - common.c, common.h (se houver)

Prazo de entrega

Os trabalhos poderão ser entregues até às 23:59 (vinte e três e cinquenta e nove) do dia especificado para a entrega. O horário de entrega deve respeitar o relógio do sistema Moodle, ou seja, a partir de 00:00 do dia seguinte à entrega no relógio do Moodle, os trabalhos **não poderão ser entregues**. Logo não serão considerados trabalhos entregues fora do prazo definido.

Dicas e Cuidados

- O guia de programação em rede do Beej (http://beej.us/guide/bgnet/) tem bons exemplos de como organizar um servidor
- Procure escrever seu código de maneira clara, com comentários pontuais e bem identado.
- Não se esqueça de conferir se seu código não possui erros de compilação ou de execução.
- Implemente o trabalho por partes. Por exemplo, implemente o tratamento das múltiplas conexões, depois crie os formatos das mensagens e, por fim, trate as mensagens no servidor ou cliente.

Exemplos de execução

Esta seção apresenta alguns exemplos de execuções do sistema. A fim de facilitar a explicação, as tabelas a seguir detalham o passo a passo dos comandos de entrada (**em negrito**) e as informações que devem ser impressas em tela em cada instante de tempo. A Tabela 1 apresenta um cenário de conexão de dois usuários (Terminal 1 e 2) com o servidor (Terminal 0).

Tempo	Terminal 0	Terminal 1	Terminal 2
t_1	**./server v6 51511**		
t_2		**./user ::1 51511**	
t_3	User 01 added		
t_4		User 01 joined the group!	
t_5			**./user ::1 51511**
t_6	User 02 added		
t_7		User 02 joined the group!	User 02 joined the group!

Tabela 1 - Cenário exemplo de abertura de conexão

A Tabela 2 apresenta um cenário que solicita os usuários conectados e o fechamento de conexão, assumindo que os usuários 01 (Terminal 1), 02 (Terminal 2) e 03 (Terminal 3) estão atualmente conectados com servidor (Terminal 0).

Tempo	Terminal 0	Terminal 1	Terminal 2	Terminal 3
t_1		**list users**		
t_2		02 03		
t_3			**close connection**	
t_4	User 02 removed			
t_5			Removed Successfully	
t_6		User 02 left the group!		User 02 left the group!
t_7				**list users**
t_8				01

Tabela 2 - Cenário exemplo de fechamento de conexão e listar usuários

A Tabela 3 apresenta um cenário de troca de mensagens com usuários existentes e inexistentes, assumindo que os usuários 01 (Terminal 1), 02 (Terminal 2) e 03 (Terminal 3) estão atualmente conectados com servidor (Terminal 0) e o usuário 04 não está conectado.

Tempo	Terminal 0	Terminal 1	Terminal 2	Terminal 3
t_1		**send to 03 “Oi sumida”**		
t_2		P [03:27] -> 03: Oi sumida		
t_3				P [03:27] 01: Oi sumida
t_4			**send to 04 “Alo?”**	
t_5	User 04 not found			
t_6			Receiver not found	
t_7			**send all “Gnt kd o 04?”**	
t_8	[03:28] 02: Gnt kd o 04?			
t_9		[03:28] 02: Gnt kd o 04?	[03:28] -> all: Gnt kd o 04?	[03:28] 02: Gnt kd o 04?

Tabela 3 - Cenário exemplo de envio de mensagens para usuários existentes e inexistentes

A Tabela 4 apresenta um cenário em que existem 15 usuários conectados simultaneamente com o servidor v4 (Terminal 0) e um novo usuário (Terminal 1) solicita abertura de conexão.

Tempo	Terminal 0	Terminal 1
t_1		**./user 127.0.0.1 51511**
t_2		User limit exceeded

Tabela 4 - Cenário exemplo de limite de usuários excedido