

RELATÓRIO DE SISTEMAS DISTRIBUÍDOS

1. 1. Introdução

O relatório em questão trata-se de explicar o funcionamento de sockets e demonstrar na prática um chat multi-usuário usando essa tecnologia na prática para melhor entendimento. Além disso, será utilizado threads para cada cliente conectado.

Para isso, devemos apresentar dois conceitos: sockets e threads. Socket é uma forma de comunicação entre duas pontas, a fonte e o destino, seja entre dois processos na mesma máquina ou na rede (que é o caso deste chat). Outro conceito, portanto, seria o de threads que vai ser gerado a cada nova conexão. Threads fazem o programa ter mais de um fluxo de execução, ou seja, o processo principal se “divide” em duas ou mais partes para poder executar outras funções independentemente. Os threads possuem grande importância, pois o servidor irá precisar lidar com as conexões pendentes, ou seja, em qualquer momento pode ter algum cliente que queira conectar, e ao mesmo tempo com os clientes já conectados para enviar e receber as mensagens.

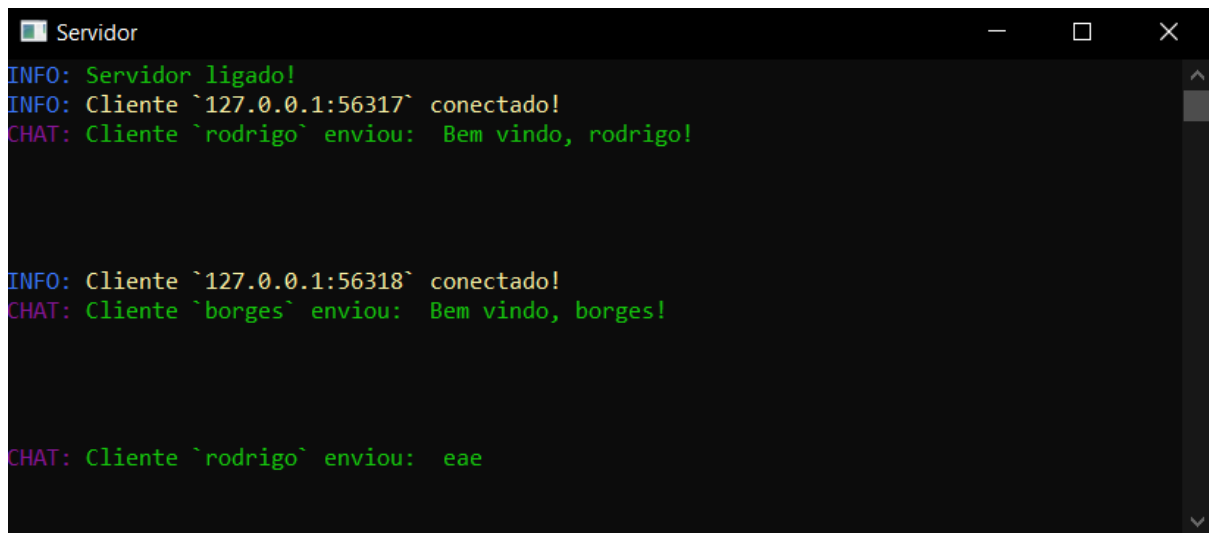
1. 2. Vídeo

O link a seguir: <https://www.youtube.com/watch?v=IRelSIDUVmc>, apresenta a explicação do código e o funcionamento na prática do cliente e do servidor.

2. 1. Aplicação

Como é possível visualizar na figura 1, o servidor é uma aplicação separada do cliente. É uma aplicação console onde ao abrir já vai iniciar o servidor automaticamente na porta 9999. Após iniciado o servidor, ele estará pronto para receber conexões e lidar com elas.

Na figura 2, é possível visualizar a interface do cliente, onde o usuário vai poder inserir um nome de usuário e então clicar em conectar. Dessa forma ao realizar esse processo, é iniciado uma comunicação TCP-IP para verificar se o servidor está aberto no IP local da máquina e na porta 9999, por exemplo. Se sim, ele conecta ao servidor e o mesmo capta a informação de conexão pendente e aceita essa conexão, ao aceitar, cria um thread e preenche um objeto chamado “Cliente” com seu socket e thread. Esse objeto é então adicionado a uma lista de clientes (informação contida no servidor), sendo muito importante para, por exemplo: se tiverem 3 pessoas conectadas, pessoa A, B e C e a pessoa A enviar uma mensagem, essa mensagem é recebida no servidor e o servidor vai pegar essa mensagem e analisar a lista de clientes conectados e percorrendo-a consegue enviar a mensagem da pessoa A para todos os outros clientes, inclusive para si.



```
INFO: Servidor ligado!  
INFO: Cliente `127.0.0.1:56317` conectado!  
CHAT: Cliente `rodrigo` enviou: Bem vindo, rodrigo!  
  
INFO: Cliente `127.0.0.1:56318` conectado!  
CHAT: Cliente `borges` enviou: Bem vindo, borges!  
  
CHAT: Cliente `rodrigo` enviou: eae
```

Figura 1 - Aplicação servidor sendo executada

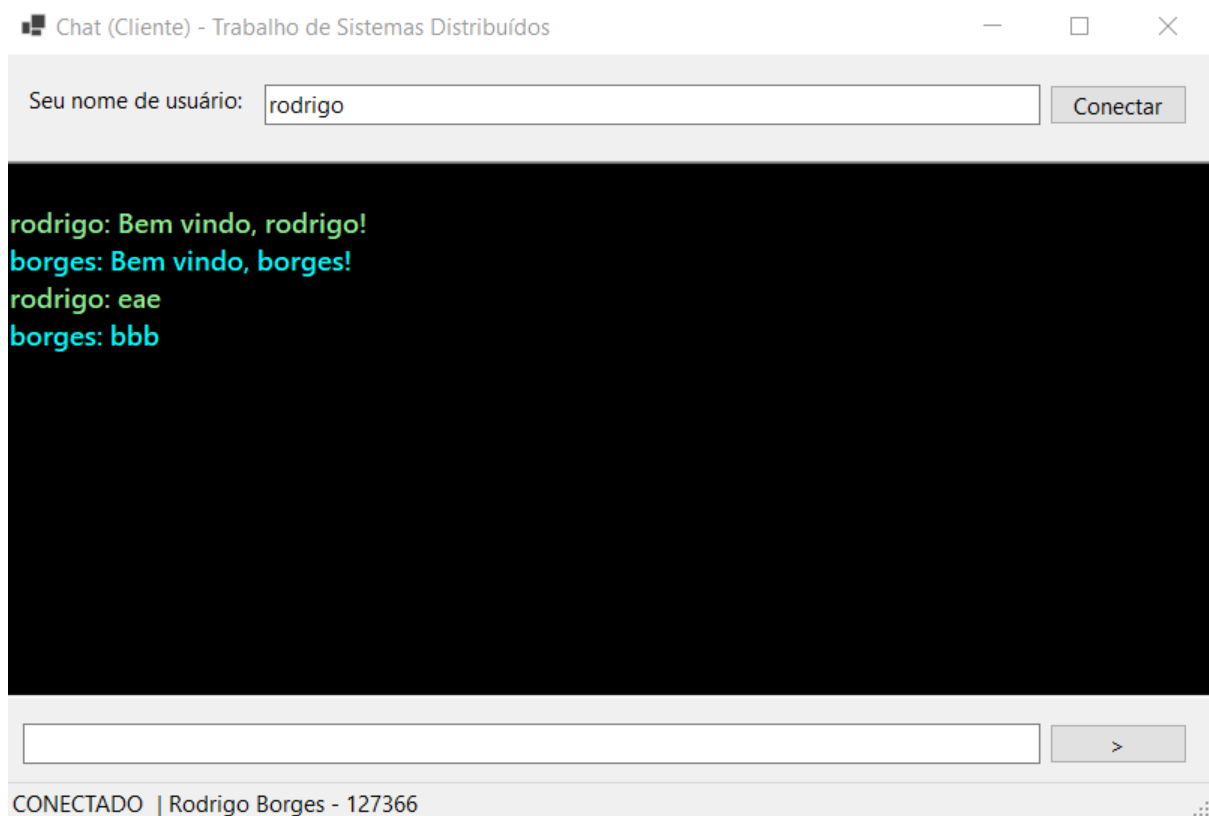


Figura 2 - Aplicação cliente sendo executada

2. 2. Instalação e execução de bibliotecas

A aplicação, tanto servidor e cliente, foram construídas utilizando a linguagem de programação C# e bibliotecas nativas da linguagem, usando o framework multiplataforma e de código aberto da Microsoft, .NET (<https://dotnet.microsoft.com/download/dotnet/3.1>).

Para executar o chat, primeiramente execute o servidor sendo o arquivo “ChatWebSocketServer.exe” e após a mensagem de início, indicada na figura 1, será possível abrir clientes. Para abrir a aplicação cliente, figura 2, basta executar o “ChatWebSocketClient.exe”.

As aplicações se encontram na pasta raiz chamada “chatwebsocket”. Basta abrir a pasta, cliente ou servidor, acessar “bin\Debug\netcoreapp3.1\” e executar o arquivo com extensão “.exe” conforme o nome, e mostrados na figura 3 de forma unida para melhor entendimento.

Nome	Data de modificação	Tipo	Tamanho
.git	31/08/2021 14:14	Pasta de arquivos	
ChatWebSocketClient	17/09/2021 15:51	Pasta de arquivos	
ChatWebSocketServer	17/09/2021 19:43	Pasta de arquivos	
.gitignore	31/08/2021 13:44	Documento de Te...	7 KB
README.md	31/08/2021 13:44	Arquivo Fonte Mar...	1 KB

chatwebsocket\ChatWebSocketClient\bin\Debug\netcoreapp3.1			
Nome	Data de modificação	Tipo	Tamanho
ChatWebSocketClient.deps.json	14/09/2021 18:39	JSON File	1 KB
ChatWebSocketClient.dll	17/09/2021 19:43	Extensão de aplica...	13 KB
ChatWebSocketClient.exe	17/09/2021 19:43	Aplicativo	171 KB
ChatWebSocketClient.pdb	17/09/2021 19:43	Program Debug D...	14 KB
ChatWebSocketClient.runtimeconfig.dev.j...	14/09/2021 18:39	JSON File	1 KB
ChatWebSocketClient.runtimeconfig.json	14/09/2021 18:39	JSON File	1 KB
ChatWebSocketServer.deps.json	17/09/2021 15:38	JSON File	1 KB
ChatWebSocketServer.dll	17/09/2021 19:43	Extensão de aplica...	9 KB
ChatWebSocketServer.exe	17/09/2021 19:43	Aplicativo	171 KB
ChatWebSocketServer.pdb	17/09/2021 19:43	Program Debug D...	12 KB
ChatWebSocketServer.runtimeconfig.dev....	17/09/2021 15:38	JSON File	1 KB
ChatWebSocketServer.runtimeconfig.json	17/09/2021 15:38	JSON File	1 KB

chatwebsocket\ChatWebSocketServer\bin\Debug\netcoreapp3.1			
Nome	Data de modificação	Tipo	Tamanho
ChatWebSocketServer.deps.json	17/09/2021 15:38	JSON File	1 KB
ChatWebSocketServer.dll	17/09/2021 19:43	Extensão de aplica...	9 KB
ChatWebSocketServer.exe	17/09/2021 19:43	Aplicativo	171 KB
ChatWebSocketServer.pdb	17/09/2021 19:43	Program Debug D...	12 KB
ChatWebSocketServer.runtimeconfig.dev....	17/09/2021 15:38	JSON File	1 KB
ChatWebSocketServer.runtimeconfig.json	17/09/2021 15:38	JSON File	1 KB

Figura 3 - Pasta raiz do projeto e seus caminhos para execução do cliente e servidor

2. 3. Código

Na classe “coração” da aplicação do servidor, Server, é onde acontece todo o gerenciamento. Em seu construtor é criada a configuração da porta, passada como parâmetro e como utilizado nesse exemplo: 9999, é criado o TcpListener e instanciado a lista de clientes conectados para posteriormente ir adicionando clientes nessa listagem conforme irem conectando.

Na figura 4, o método “Start” pega o TcpListener anteriormente configurado e inicia. Com isso o servidor já está rodando, após é criado um loop infinito para poder ficar sempre aguardando possíveis novas conexões. O método “AcceptSocket” aceita a requisição da conexão pendente e retorna um objeto do tipo Socket, onde após, é criado um thread para poder processar essa conexão e é enviado como parâmetro esse socket retornado. A criação de threads para cada conexão do cliente é importante visto que o servidor tem que lidar com novas conexões, as atuais e também o envio e recebimento de mensagens, para isso há a necessidade mais fluxos de execução, ou seja, threads.

```
//Método responsável por iniciar o servidor após ter definido as configurações iniciais no construtor
1 referência
public void Start()
{
    try
    {
        if(_serverListener == null)
        {
            ConsoleLog.Write("TcpListener está nulo!", ConsoleColor.Green, MessageType.ERROR);
            return;
        }

        _serverListener.Start();

        ConsoleLog.Write("Servidor ligado!", ConsoleColor.Green, MessageType.INFO);

        while (true)
        {
            //Aceita uma conexão pendente e cria um thread para o cliente atual
            Socket clientSocket = _serverListener.AcceptSocket();
            _clientThread = new Thread(ProcessClientConnection);
            _clientThread.IsBackground = true;
            _clientThread.Start(clientSocket);
        }
    }
    catch(Exception ex)
    {
        ConsoleLog.Write(ex.Message, ConsoleColor.Green, MessageType.ERROR);
    }
}
```

Figura 4 - Função Start da classe Server

Na figura 5, é possível ver o método responsável por processar a conexão do cliente onde vai criar um objeto do tipo Cliente (classe contendo informações básicas do usuário) passando como parâmetro o thread atual do cliente e seu socket, e adicionar na lista de clientes conectados. Essa lista vai ser importante para utilização do método “Broadcast”, como visto na figura 6, que justamente percorre essa lista e envia mensagem para todos os clientes conectados. Esse método chama como base outro método para auxílio, o “SendToClient”, que vai pegar o cliente individual e enviar uma mensagem, a mensagem é convertida de string para um vetor de bytes e enviada via stream.

```

private void ProcessClientConnection(object socket)
{
    Socket clientSocket = (Socket)socket;
    bool keepAlive = true;

    ConsoleLog.Write(string.Format("Cliente `{0}` conectado!", clientSocket.RemoteEndPoint), ConsoleColor.Yellow, MessageType.INFO);

    Client clientConnected = new Client(clientSocket, _clientThread);
    _connectedClients.Add(clientConnected);

    while (keepAlive)
    {
        try
        {
            byte[] buffer = new byte[1024];
            //Recebe dados e coloca dentro da variável buffer
            int bytesReceived = clientSocket.Receive(buffer);

            if (bytesReceived > 0)
            {
                string clientMessage = Encoding.UTF8.GetString(buffer);
                string[] clientInfo = clientMessage.Split(':');

                string username = clientInfo[0];
                string message = clientInfo[1];

                ConsoleLog.Write(string.Format("Cliente `{0}` enviou: {1}", username, message), ConsoleColor.Green, MessageType.CHAT);

                Broadcast(clientMessage);

                if (message.Equals("quit"))
                {
                    DisconnectClient(clientConnected);
                    keepAlive = false;
                }
            }
        }
        catch (Exception ex)
        {
            ConsoleLog.Write(ex.Message, ConsoleColor.Green, MessageType.ERROR);
            DisconnectClient(clientConnected);
        }
    }
}

```

Figura 5 - Função *ProcessClientConnection* da classe *Server*

```

//Método responsável por enviar mensagem para o cliente especificado
1 referência
private void SendToClient(Client client, string message)
{
    try
    {
        //Converte string para um array de bytes
        byte[] buffer = Encoding.UTF8.GetBytes(message);
        client.Socket.Send(buffer, buffer.Length, 0);
    }
    catch (Exception ex)
    {
        ConsoleLog.Write(ex.Message, ConsoleColor.Green, MessageType.ERROR);
        DisconnectClient(client);
    }
}

//Método responsável por enviar mensagem para todos os clientes conectados
1 referência
private void Broadcast(string message)
{
    try
    {
        foreach (Client client in _connectedClients)
        {
            //Se por algum problema o cliente não estiver conectado não tenta enviar a mensagem para ele e passa pro próximo da lista
            if (!client.Socket.Connected)
            {
                DisconnectClient(client);
                continue;
            }

            SendToClient(client, message);
        }
    }
    catch (Exception ex)
    {
        ConsoleLog.Write(ex.Message, ConsoleColor.Green, MessageType.ERROR);
    }
}

```

Figura 6 - Funções SendToClient e Broadcast

A aplicação cliente funciona de forma semelhante. Ao invés de usar TcpListener, usa um TcpClient informando um IP e uma porta para o servidor já criado. Ao definir um nome de usuário e clicar em conectar, a conexão é feita e criado um thread para recebimento das mensagens para que não trave o fluxo atual da aplicação. Esse método fica constantemente verificando se tem algo no buffer, caso tenha, lê e converte os bytes recebidos para uma string, como é possível verificar na figura 7.

```

//Método responsável por ficar lendo o servidor e colocando na tela as mensagens
1 referência
private void ChatMessages()
{
    bool keepAlive = true;

    try
    {
        while (keepAlive)
        {
            byte[] buffer = new byte[2048];
            _networkStream.Read(buffer, 0, buffer.Length);

            string message = Environment.NewLine + Encoding.UTF8.GetString(buffer);

            //Essa condição serve para mostrar a cor "verde" para quando a mensagem for sua e "ciano" quando for de outros usuários
            if (message.Split(':')[0].Trim().ToLower().Equals(_localUsername.Trim().ToLower()))
            {
                ChatLogAppendText(message, Color.LightGreen);
            }
            else
            {
                ChatLogAppendText(message, Color.Cyan);
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

Figura 7 - Função da aplicação cliente: ChatMessages

3. Conclusão e dificuldades encontradas

Conclui-se com este trabalho a possibilidade de ter entendido na prática a utilização dos sockets juntamente com threads. Isso é de veras importante, visto que muitas vezes acaba-se ficando apenas na teoria.

As dificuldades encontradas foram encontrar conteúdos na internet que fossem realmente aproveitáveis, visto que diversos tutoriais ensinavam a fazer a mesma coisa mas de maneiras diferentes e com pouca explicação. Acaba que com tudo isso, foi feito várias vezes o chat até consolidar bem o que cada função fazia. Alguns tutoriais abordavam apenas um chat simples usando socket mas permitindo apenas o envio de uma mensagem simples e fechamento da conexão após, outros ensinavam a fazer apenas o servidor, outros ensinavam a fazer apenas o cliente. Após entender o procedimento de criação de uma conexão, foi possível vincular isso com a criação de threads para cada cliente conectado para poder enviar e receber mensagens. Outra dificuldade encontrada foi na desconexão do cliente, pois ao fechar a aplicação é enviado pro servidor uma mensagem "quit" e o servidor pega essa informação e desconecta o socket do cliente e remove esse cliente da lista de conectados, mas acaba gerando um erro no console, mas a aplicação continua rodando.