Desenvolvimento e Análise de um Protocolo de Chat Cliente-Servidor com Sockets em Python

Miguel de Souza Rocha - 241022414
John Peter Henry Jacobs - 222005887
Igor Araújo Rodrigues - 242001517

Universidade de Brasília, Depto. Ciências da Computação, Brasília - DF
CIC 01245 - Redes de Computadores - Turma 01

Palavras-chave: Sockets, TCP, Protocolo de Aplicação, Arquitetura Cliente-Servidor

Resumo

Este relatório descreve o desenvolvimento de uma aplicação de chat multiusuário na arquitetura Cliente-Servidor, utilizando Python e sockets TCP, como requisito da disciplina de Redes de Computadores. Para formalizar a comunicação, foi implementado um protocolo de aplicação customizado que adota o padrão de códigos de status HTTP. O projeto inclui mecanismos de sincronismo (*Locks*) para gerenciar a concorrência no servidor. O objetivo final é a análise detalhada desse protocolo em uma rede privada, utilizando a ferramenta Wireshark para inspecionar o envio, recebimento e encapsulamento dos pacotes.

Abstract

This report describes the development of a multi-user chat application based on the Client-Server architecture, utilizing Python and TCP sockets, as a requirement for the Computer Networks course. To formalize communication, a custom application protocol was implemented that adopts the HTTP status code standard. The project includes synchronization mechanisms (Locks) to manage concurrency on the server. The final objective is the detailed analysis of this protocol on a private network, using the Wireshark tool to inspect the sending, receiving, and encapsulation of packages..

1. Introdução

A camada de aplicação e o modelo Cliente-Servidor são fundamentais para a arquitetura da Internet. Este projeto visa aprofundar essa compreensão através da criação de uma aplicação de rede funcional, permitindo "ver os protocolos em ação" e observar o fluxo de mensagens trocadas entre as entidades.

O trabalho consiste em:

1. Desenvolver uma aplicação de chat (arquitetura Cliente-Servidor) utilizando a linguagem Python e a biblioteca socket..
2. Definir e implementar um protocolo de aplicação customizado para gerenciar a troca de mensagens e comandos (inspirado em padrões HTTP para controle de status).
3. Executar a aplicação em uma rede privada com um host servidor e múltiplos hosts clientes.
4. Utilizar o software **Wireshark** para capturar, verificar e analisar os pacotes trocados, validando a operação do protocolo e o encapsulamento.

A seguir, a Seção 2 descreve a concepção da solução e a Seção 3 apresenta o ambiente experimental e a análise dos resultados obtidos.

2. Fundamentação Teórica

2.1. Arquitetura Cliente-Servidor e Camadas de Protocolo

O projeto adota a arquitetura **Cliente-Servidor**, na qual o server.py atua como um host central que oferece o serviço de chat, e os client.py são os hosts que solicitam e consomem esse serviço.

O protocolo de transporte escolhido foi o **TCP (Transmission Control Protocol)** (socket de fluxo), pois garante a entrega confiável e ordenada das mensagens de chat, sendo crucial para a integridade da comunicação.

2.2. Protocolo de Aplicação Customizado

Para o chat, foi definido o seguinte protocolo customizado sobre TCP:

Tipo de Mensagem	**Exemplo de Envio do Cliente**	**Formato da Resposta do Servidor**	**Finalidade**
Autenticação	[NICKNAME]	200 OK - Connected	Confirma a entrada do usuário.

Mensagem	MSG <texto>	200 OK	Envio de mensagem para *broadcast*.
Comando	LIST	200 OK - User List	Requisição para listar usuários.
Saída	QUIT	200 OK - Disconnecting	Encerramento da sessão.
Erro	Comando inválido	404 Not Found / 400 Bad Request	Indica falha na requisição.

2.3. Sincronismo e Deadlocks (Multithreading)

O servidor precisa lidar com a concorrência de múltiplos clientes (handle_client é executada em threads separadas). Para prevenir **Condições de Corrida** ao acessar as listas globais (clientes e nomes_usuarios), foi utilizado o mecanismo de **Lock (Mutex)** do Python.

A **Seção Crítica** (código onde as variáveis compartilhadas são modificadas) é protegida por lock.acquire() e lock.release(), garantindo o **sincronismo** dos dados e evitando inconsistências que poderiam levar a deadlocks (bloqueio mútuo) se a lógica de acesso aos recursos não fosse gerenciada corretamente.

2.4. Interface de Usuário (Terminal CLI)

A interface de usuário da aplicação (client.py) foi desenvolvida como uma interface de linha de comando (*Command Line Interface* - CLI), maximizando a portabilidade e a compatibilidade com o ambiente de terminal (terminal.exe no Windows, bash/zsh no Linux/macOS). A interação é baseada em texto e segue um modelo simples de **Entrada/Saída Concorrente**.

1. **Entrada do Usuário:** A thread principal do cliente é dedicada a exibir o *prompt* (>) e capturar a entrada do usuário (input()).
2. **Saída do Servidor (Notificações):** Uma thread secundária (receive_messages) é responsável por receber mensagens e notificações do servidor. Para garantir que as mensagens recebidas não interfiram na linha de digitação do usuário, a saída utiliza sys.stdout.write e sys.stdout.flush.

Formato da Interação:

Tipo de Interação	Exemplo no Terminal	Descrição
Inicialização	Escolha seu nickname: Dudinha	Solicitação inicial de identificação do cliente.
Comando de Chat	> Olá a todos!	O cliente prefixa a mensagem com MSG e a envia ao servidor.
Comando de Controle	> /list	O cliente envia o comando LIST ao servidor.
Resposta de Sucesso	[200 OK] OK - Mensagem enviada com sucesso.	Feedback do servidor após uma ação bem-sucedida (ex: envio de mensagem).
Resposta de Erro	[404 Not Found] Comando desconhecido: TESTE	Feedback do servidor para requisições com sintaxe inválida ou comandos inexistentes.
Broadcast (Mensagem)	[João]: Olá!	Mensagem recebida de outro cliente, retransmitida pelo servidor.

A interface foi projetada para ser **minimalista** e focar na troca de comandos e na validação das respostas de protocolo, conforme os requisitos de desenvolvimento do protocolo de aplicação.

3. Ambiente Experimental e Análise de Resultados

3.1. Descrição do Cenário

Hardware e Software:

- **Host Servidor:** Notebook/Desktop rodando Python 3.x e server.py.
- **Hosts Clientes (Mínimo 2):** Notebooks/Smartphones rodando Python 3.x e client.py.
- **Topologia de Rede:** Rede privada WLAN (Wi-Fi) ou cabeada, com todos os hosts na mesma sub-rede.
- **Ferramenta de Análise:** Wireshark.

- **Configurações:** O IP do servidor foi obtido via ipconfig (Windows) ou ifconfig (Linux/Unix) e configurado na variável HOST dos clientes.

3.2. Análise de Resultados (Quadro 1)

O Wireshark foi inicializado e a captura foi realizada na interface de rede utilizada. Os clientes executaram os comandos (NICK, MSG, LIST, QUIT) enquanto o tráfego era monitorado.

A. Identificação da versão ou tipo da aplicação no servidor

Resposta: A versão ou tipo da aplicação **não** é identificada em um campo padrão do cabeçalho TCP ou IP, mas sim na **Camada de Aplicação** (carga útil) do nosso protocolo customizado.

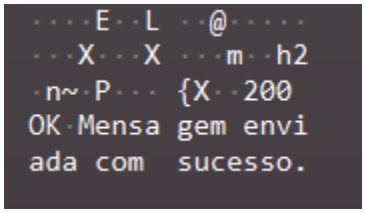


Imagem 1 - Tela do Pacote no WireShark

Justificativa: A informação está presente no corpo do pacote (carga útil), onde o servidor envia respostas como: 200 OK - Connected ou 200 OK - User List.

B. Endereços IP (Clientes e Servidor)

Resposta:

- **Endereço IP do Servidor: 192.168.0.88**
- **Endereço IP do Cliente 1: 192.168.0.88**
- **Endereço IP do Cliente 2: 192.168.0.20**

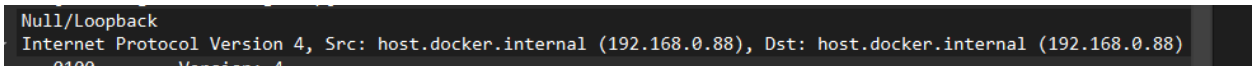


Imagem 2 - Captura de tela do Wireshark

Justificativa: Os IPs são identificados nos campos **Source** (Origem) e **Destination** (Destino) do **Cabeçalho IP** (Camada de Rede).

C. Protocolo de Transporte (TCP ou UDP)

Resposta: O protocolo de transporte utilizado é o **TCP (Transmission Control Protocol)**.

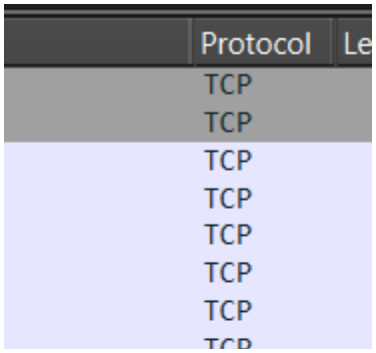


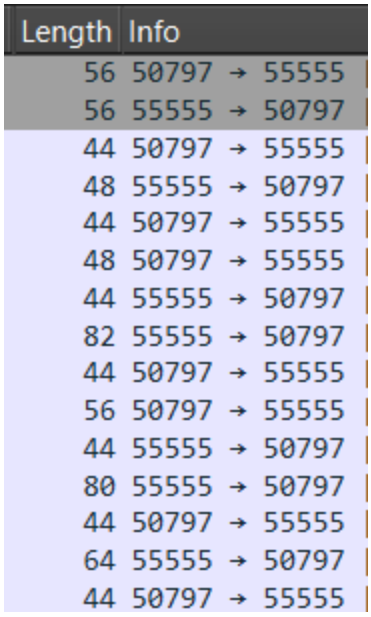
Imagem 3 - Captura de Tela no WireShark

Justificativa: A escolha do TCP foi intencional para garantir a **confiabilidade** e a **ordem de entrega** das mensagens de chat. A confirmação é observada pela sequência de pacotes SYN, SYN-ACK, ACK para estabelecer a conexão e o uso de números de sequência/ACK nos dados subsequentes. Esta informação é visível no campo *Protocol* da lista de pacotes e no detalhe do **Cabeçalho TCP**.

D. Portas (Destino e Origem do Cliente)

Resposta:

- **Porta de Destino do Cliente:** 55555
- **Porta de Origem do Cliente 1:** 50797
- **Porta de Origem do Cliente 2:** 53818



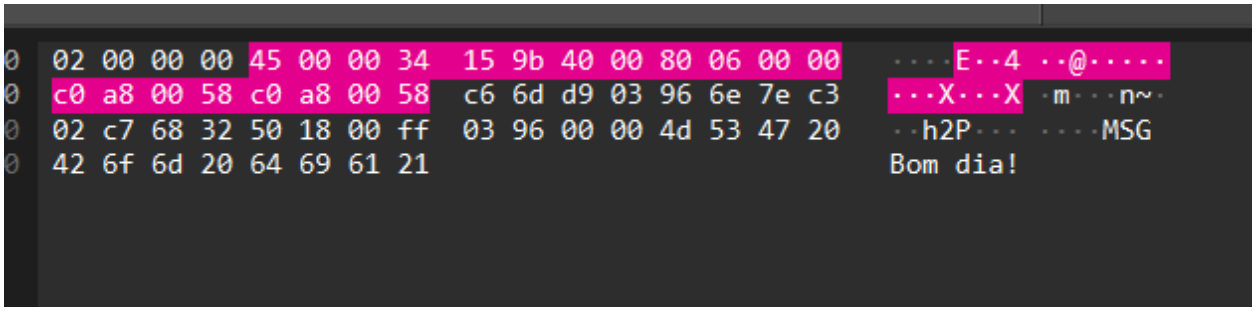
Length	Info
56	50797 → 55555
56	55555 → 50797
44	50797 → 55555
48	55555 → 50797
44	50797 → 55555
48	50797 → 55555
44	55555 → 50797
82	55555 → 50797
44	50797 → 55555
56	50797 → 55555
44	55555 → 50797
80	55555 → 50797
44	50797 → 55555
64	55555 → 50797
44	50797 → 55555

Imagem 4 - Captura de Tela no WireShark

Justificativa: As portas são encontradas no **Cabeçalho TCP** (Camada de Transporte). A **Porta de Destino** é sempre a porta conhecida do servidor (55555). A **Porta de Origem** é uma porta alta e efêmera escolhida aleatoriamente pelo sistema operacional do cliente.

E. Carga Útil e Conformidade

Resposta: Sim, a carga útil corresponde exatamente ao que é esperado pelo nosso protocolo.



```
0  02 00 00 00 45 00 00 34  15 9b 40 00 80 06 00 00   ····E··4 ··@·····
0  c0 a8 00 58 c0 a8 00 58  c6 6d d9 03 96 6e 7e c3   ···X···X ·m···n~·
0  02 c7 68 32 50 18 00 ff  03 96 00 00 4d 53 47 20   ··h2P··· ····MSG
0  42 6f 6d 20 64 69 61 21                           Bom dia!
```

Imagem 5 - Captura de Tela no WireShark

Justificativa:

- **Exemplo Cliente (Requisição):** A carga útil contém a string do comando: *[MSG] Bom dia!*
- **Exemplo Servidor (Resposta):** A carga útil contém o status e a mensagem de protocolo*: 200 OK: Mensagem enviada com sucesso.*

A inspeção da carga útil (camada mais interna do pacote) confirma que a aplicação está transmitindo as strings de texto conforme o formato do protocolo de aplicação customizado que foi definido e implementado.

F. Inspeção de Dados Brutos e Encapsulamento

Analogia: A janela de dados brutos demonstra o princípio do **Encapsulamento**. Observa-se que a **Carga Útil** (dados da Camada de Aplicação: *MSG texto*) é encapsulada sequencialmente:

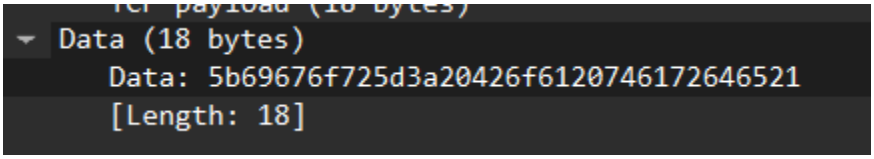


Imagem 6 - Captura de Tela no WireShark

1. **Cabeçalho TCP:** Adiciona as informações de porta, sequência e ACK (Camada de Transporte).
2. **Cabeçalho IP:** Adiciona os endereços IP de origem e destino (Camada de Rede).
3. **Cabeçalho Ethernet:** Adiciona os endereços MAC e o Frame Check Sequence (Camada de Enlace).

Este empilhamento de cabeçalhos nos dados brutos confirma o modelo de camadas estudado, onde a informação da camada superior é o *payload* da camada inferior.

4. Conclusões

O projeto foi bem-sucedido ao demonstrar a viabilidade de construir um protocolo de aplicação customizado sobre sockets TCP. A implementação do *multithreading* com *Locks* no servidor atendeu ao requisito de sincronismo, garantindo a integridade dos dados compartilhados. A análise com o Wireshark permitiu verificar, de forma empírica, o funcionamento das camadas de rede (IP), transporte (TCP) e a fidelidade da carga útil (protocolo de aplicação) conforme o modelo de encapsulamento teórico.

O código fonte completo dos arquivos server.py e client.py está disponível no seguinte link:

Link para o Repositório GitHub: https://github.com/Igualous/Python-Socket-Chat

5. Bibliografia

1. **Material da disciplina disponível na plataforma Aprender 3**
2. **Repositório no GitHub sobre programação com sockets: https://github.com/Gabrielcarvfer/Redes-de-Computadores-UnB/**

Link para o Vídeo de Demonstração: https://youtu.be/_WTYvWWCOUw