Trabalho Prático: Implementação e Análise TCP vs. TLS

Manoel Silva, Daniel Arruda

Resumo

Este trabalho apresenta um estudo comparativo entre transmissões de arquivos utilizando conexões TCP convencionais e conexões seguras com TLS (Transport Layer Security). Foi desenvolvido um sistema cliente-servidor em Python capaz de transmitir arquivos de texto através de ambos os protocolos. Os experimentos incluem capturas de pacotes de rede utilizando Wireshark, permitindo análise detalhada das diferenças entre comunicação em texto plano e comunicação criptografada. Os resultados demonstram o overhead introduzido pelo TLS em termos de tamanho de pacotes e tempo de transmissão, enquanto evidenciam os ganhos significativos em segurança e confidencialidade dos dados transmitidos.

1 Introdução

A segurança na transmissão de dados tornou-se um requisito fundamental em aplicações modernas de rede. Com o crescimento exponencial de ameaças cibernéticas e interceptações maliciosas, a proteção de informações durante o tráfego em redes públicas é essencial para garantir confidencialidade, integridade e autenticidade dos dados.

O protocolo TCP (Transmission Control Protocol) fornece comunicação confiável entre sistemas, garantindo entrega ordenada de pacotes e controle de fluxo. Entretanto, TCP por si só não oferece nenhuma forma de criptografia, transmitindo dados em texto plano e tornando-os vulneráveis a ataques de interceptação (man-in-the-middle) e espionagem.

O TLS (Transport Layer Security), sucessor do SSL (Secure Sockets Layer), é um protocolo criptográfico projetado para fornecer comunicação segura sobre redes de computadores. TLS opera sobre TCP, adicionando camadas de segurança através de criptografia, autenticação de endpoints e verificação de integridade de mensagens.

Este trabalho tem como objetivo implementar e comparar sistemas de transmissão de arquivos utilizando TCP puro e TCP com TLS, analisando aspectos de segurança, performance e overhead introduzido pela camada de criptografia.

2 Transport Control Protocol

O TCP (Transmission Control Protocol) é um protocolo da camada de transporte responsável por fornecer comunicação confiável entre dois hosts. Ele estabelece uma conexão antes que qualquer dado seja transmitido, utilizando o processo conhecido como three-way handshake.

Nesse procedimento, o cliente envia um segmento SYN, o servidor responde com SYN-ACK, e o cliente finaliza com ACK, formando assim um canal lógico entre as partes.

Após estabelecida a conexão, o TCP garante que os dados sejam entregues de forma ordenada, sem perdas e sem duplicações, utilizando mecanismos como numeração de bytes, confirmações de recebimento (ACKs) e retransmissões em caso de falhas. O protocolo também controla o fluxo e a congestão da rede, ajustando dinamicamente a taxa de envio para evitar sobrecargas e garantir desempenho estável.

Por fim, a conexão é encerrada mediante outro processo de troca de sinais, geralmente envolvendo quatro mensagens (FIN e ACK), garantindo que ambos os lados encerrem a comunicação de forma coordenada.

Contudo, tal protocolo não garante a segurança dos dados trafegados, sendo esta uma comunicação em texto plano. Por conta disso, o TLS foi introduzido para garantir a confidencialidade e a segurança do tráfego desses dados pela rede.

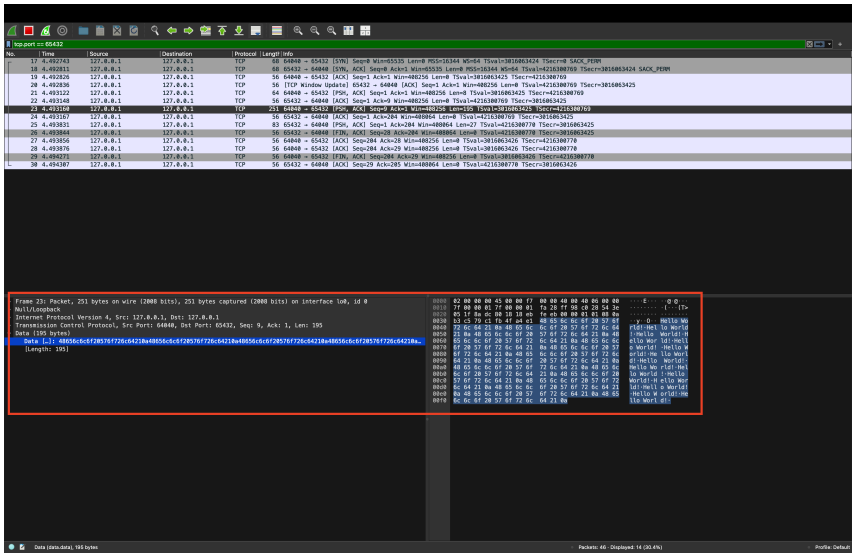

Figura 1: Texto Plano TCP do Cliente

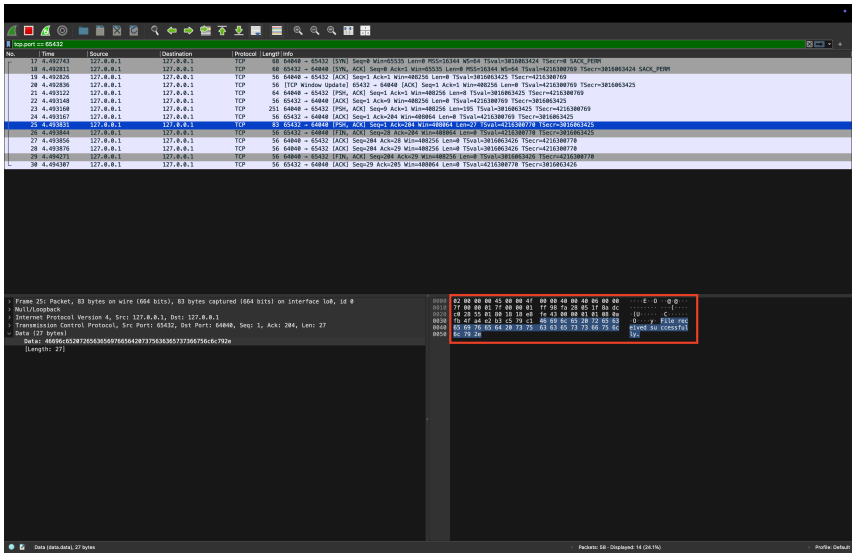

Figura 2: Texto Plano TCP do Servidor

3 Transport Layer Security

Diferente o TCP, o TLS (Transport Layer Security) é um protocolo utilizado para prover segurança na comunicação entre cliente e servidor, operando sobre o TCP. Embora resida tecnicamente na camada de aplicação, do ponto de vista do desenvolvedor, ele é um protocolo na camada de transporte.

Seu funcionamento se divide em duas etapas principais: o handshake e a criptografia dos dados transmitidos.

O handshake TLS é a sequência de mensagens que estabelece parâmetros de segurança entre cliente e servidor antes da transmissão de dados de aplicação. Seu objetivo é: (1) autenticar o servidor (e opcionalmente o cliente), (2) negociar algoritmos (cipher suite), (3) estabelecer segredos compartilhados (chaves de sessão) e (4) verificar que ambas as partes possuem os mesmos segredos (mensagens `Finished`). Após o handshake, o Record Protocol protege qualquer `application_data` por cifragem autenticada (AEAD) ou por combinação MAC+cifra dependendo da versão.

3.1 Funcionamento do Handshake TLS

O processo pode ser dividido em quatro fases principais.

3.1.1 Fase 1: *ClientHello* e *ServerHello*

Nesta fase, cliente e servidor negociam os parâmetros iniciais da conexão segura.

ClientHello O cliente inicia a comunicação enviando:

- **Versão suportada do TLS**: indica a versão máxima do protocolo que pode utilizar.
- **Cipher Suites**: lista ordenada dos conjuntos criptográficos suportados.
- **Client Random**: valor aleatório que servirá de insumo para a derivação das chaves.
- **Session ID**: usado para tentar retomar uma sessão anterior ou indicar que será necessário um handshake completo.

ServerHello O servidor responde escolhendo os parâmetros da sessão:

- Seleciona a **cipher suite** que será usada.
- Envia o **Server Random**.
- Retorna o **Session ID**, confirmando a negociação.

3.1.2 Fase 2: Autenticação do Servidor

Após o *ServerHello*, o servidor envia as mensagens responsáveis por autenticar sua identidade e definir os parâmetros de troca de chaves.

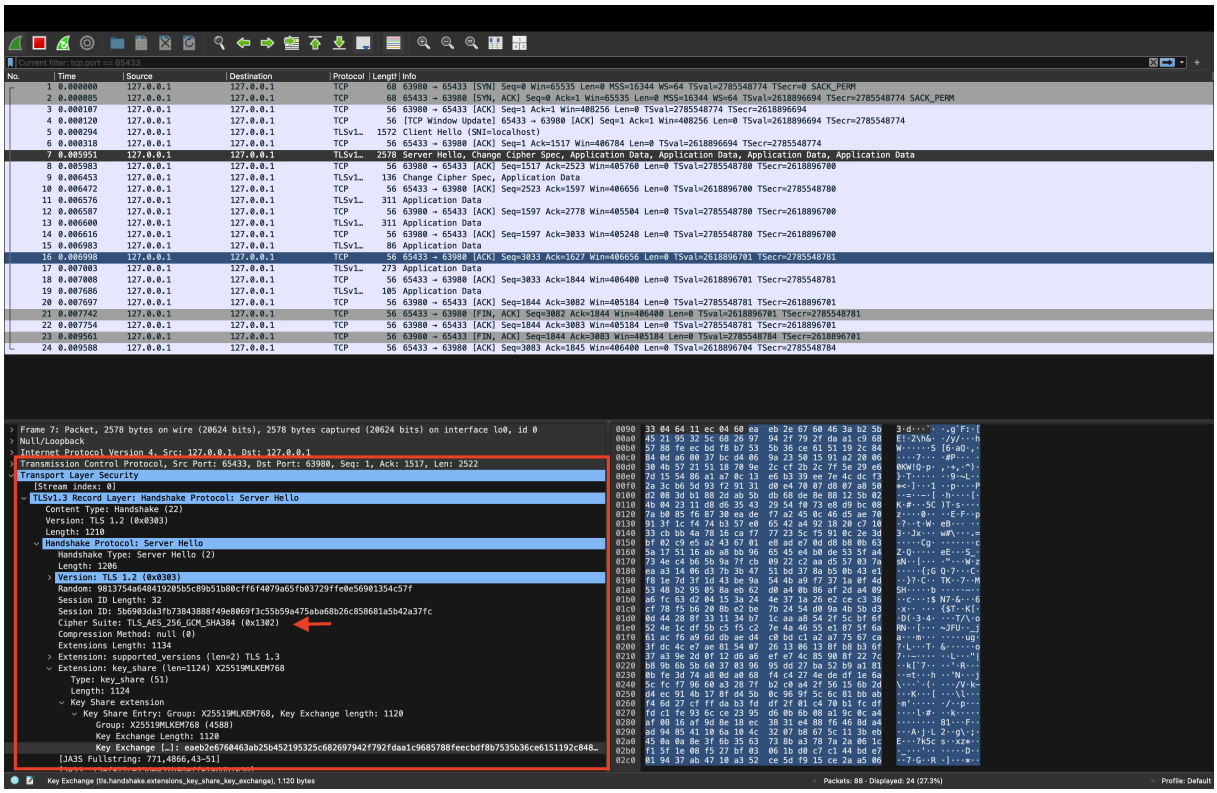

Figura 3: ServerHello

- **Certificate**: contém a chave pública e identifica o servidor.
- **Server Key Exchange**
- **CertificateRequest** (opcional): solicita certificado do cliente.
- **ServerHelloDone**: indica que o servidor concluiu sua parte inicial do handshake.

3.1.3 Fase 3: Resposta do Cliente

O cliente agora valida as informações do servidor e envia seus próprios parâmetros para finalizar a troca de chaves.

- **Validação do certificado**: o cliente verifica autenticidade, validade e cadeia de confiança.
- **ClientKeyExchange**:
- **Certificate** e **CertificateVerify** (se solicitados): autenticam o cliente.

3.1.4 Fase 4: Finalização da Negociação

Uma vez derivadas as chaves simétricas, ambas as partes ativam a criptografia.

Mensagens do cliente

- **ChangeCipherSpec**: informa que o cliente começará a usar as chaves negociadas.
- **Finished**: primeira mensagem já cifrada, confirmando a integridade do handshake.

Mensagens do servidor

- **ChangeCipherSpec**: o servidor também passa a usar as chaves derivadas.
- **Finished**: confirma que a negociação foi concluída com sucesso.

Após a troca dessas mensagens, o handshake está finalizado e o canal seguro é estabelecido.

3.2 Criptografia no TLS

A criptografia no TLS ocorre na *Record Layer*, que utiliza as chaves derivadas durante o handshake para proteger a comunicação.

3.2.1 Derivação das Chaves

O TLS utiliza três valores:

- **Client Random**
- **Server Random**
- **Pre-Master Secret** (obtido via ECDHE ou RSA)

Esses valores são combinados em uma função de derivação de chaves (*HKDF* nas versões modernas) para produzir:

- Chave de criptografia do cliente.
- Chave de criptografia do servidor.
- Chaves de autenticação e integridade.

3.2.2 Proteção dos Dados

Cada mensagem enviada é tratada na Record Layer:

- É autenticada (HMAC ou AEAD, dependendo da versão do TLS).
- É criptografada com a chave da direção correspondente.
- É encapsulada em um *TLS Record* e transmitida.

Essa separação garante confidencialidade, integridade e proteção contra ataques de replay.

Assim, o TLS acrescenta uma camada de segurança ao canal confiável já estabelecido pelo TCP, tornando a comunicação adequada para aplicações sensíveis, como transações bancárias, autenticação e transmissão de informações privadas.

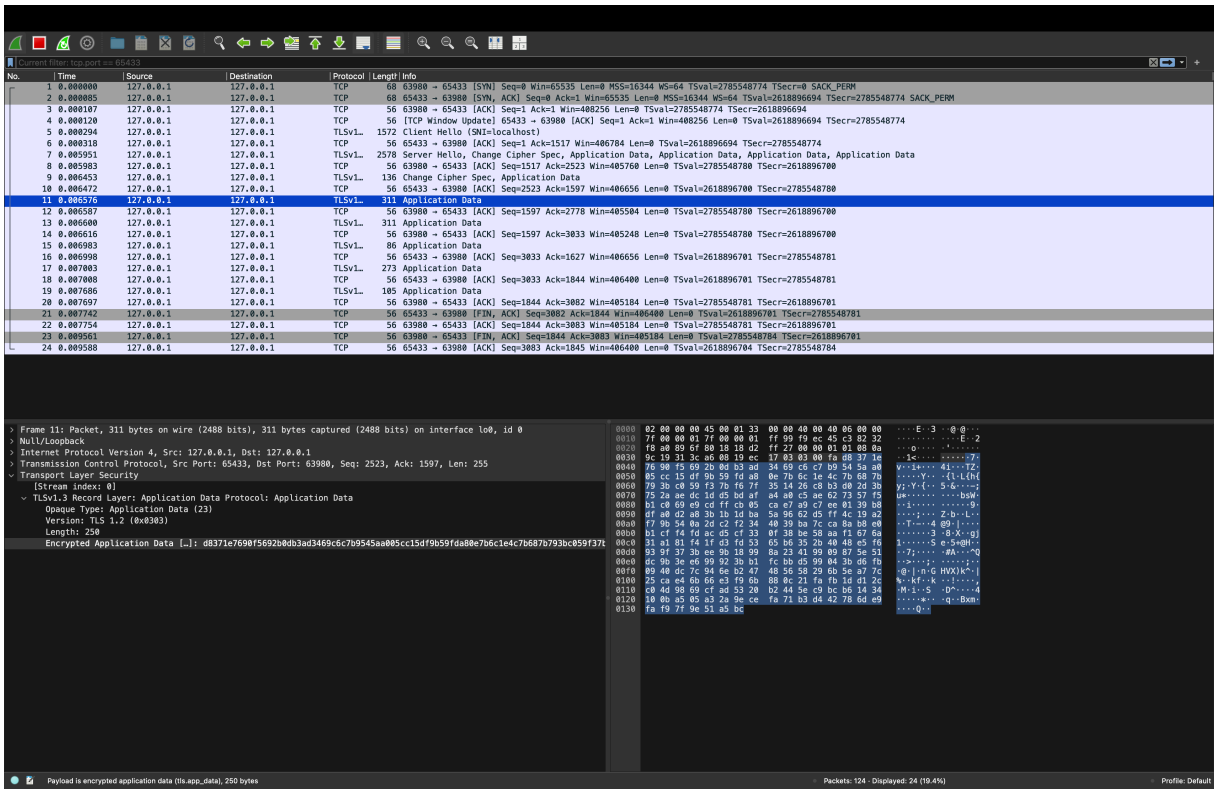

Figura 4: Texto Cifrado no TLS

4 Implementação

4.1 Server

```python
class Server:
    def __init__(self, host, port, use_tls):
        self.host = host
        self.port = port
        self.use_tls = use_tls

    def start(self):
        os.makedirs(FILE_SAVE_PATH, exist_ok=True) # Ensure the
            directory for saving files exists

        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM
            )
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
             1)
        server_socket.bind((self.host, self.port))
        server_socket.listen(5) # Allow up to 5 queued connections

```

```
        print(f"Server listening on {self.host}:{self.port} {'with TLS' if self.use_tls else 'without TLS'}")

        context = None
        if self.use_tls:
            context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
            context.load_cert_chain(certfile='server.crt', keyfile='server.key')
        try:
            while True:
                conn, addr = server_socket.accept()
                if self.use_tls:
                    conn = context.wrap_socket(conn, server_side=True) # Wrap accepted socket with TLS
                client_thread = threading.Thread(target=self.handle_client, args=(conn, addr)) # Handle each client in a new thread
                client_thread.start()
        except KeyboardInterrupt:
            print("Server shutting down.")
        finally:
            server_socket.close()

    def handle_client(self, conn, addr):
        print(f"Connection from {addr} has been established.")
        start_time = time.time()
        total_data_received = 0

        data_chunks = []
        try:
            # First, receive the file size (8 bytes) - ensure we get exactly 8 bytes
            size_data = b''
            while len(size_data) < 8:
                chunk = conn.recv(8 - len(size_data))
                if not chunk:
                    print(f"Connection closed while receiving header from {addr}")
                    return
                size_data += chunk

            expected_size = int.from_bytes(size_data, byteorder='big')
            print(f"Expecting {expected_size} bytes from {addr}")

            # Now receive exactly that many bytes
            while total_data_received < expected_size:
                remaining = expected_size - total_data_received
                chunk_size = min(BUFFER_SIZE, remaining)
                try:
                    data = conn.recv(chunk_size)
                    if not data:
                        break
                    total_data_received += len(data)
                    data_chunks.append(data)
                except (ConnectionResetError, BrokenPipeError, ssl.SSLError) as recv_error:
```

```
                    print(f"Error receiving data from {addr}: {
                        recv_error}")
                    break

            data = b''.join(data_chunks)
            end_time = time.time()
            duration = end_time - start_time

            if total_data_received > 0:
                print(f"Received {total_data_received} bytes from {addr}
                    in {duration:.6f} seconds.")
                #timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
                #self.save_received_file(data, f"received_from_{addr[0]}
                    _{addr[1]}_{timestamp}.bin")

            # Send acknowledgment
            try:
                ack_message = "File received successfully.".encode('utf
                    -8')
                conn.sendall(ack_message)
            except (ConnectionResetError, BrokenPipeError, ssl.SSLError)
                as send_error:
                print(f"Error sending acknowledgment to {addr}: {
                    send_error}")

            print(f"Connection from {addr} closed. Received {
                total_data_received} bytes in {duration:.6f} seconds.")
        except Exception as e:
            print(f"Unexpected error from {addr}: {e}")
        finally:
            conn.close()
```

Listing 1: Servidor

4.2 Client

```
class Client:
    def __init__(self, host, port, use_tls):
        self.host = host
        self.port = port
        self.use_tls = use_tls
        self.stats = {'data_size': 0,
                      'transfer_time': 0.0,
                      'average_speed': 0.0,
                      'connection_type': 'TLS' if use_tls else 'TCP',
                      'timestamp': '',}

    def connect(self):
        try:
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM
                )
            if self.use_tls:
                context = ssl.create_default_context()
                if DEBUG:
                    context.check_hostname = False
                    context.verify_mode = ssl.CERT_NONE # Disable
                        certificate verification for debugging to accept
```

```
                    self-signed certs
                self.sock = context.wrap_socket(self.sock,
                    server_hostname=self.host)

            self.sock.connect((self.host, self.port))
            print(f"Connected to server {self.host}:{self.port} {'with
                TLS' if self.use_tls else 'without TLS'}.")
            if self.use_tls:
                print(f"Server certificate:\n{self.sock.getpeercert()}")
                print(f"TLS version: {self.sock.version()}")
                print(f"Cipher: {self.sock.cipher()}")
                print(f"Compression: {self.sock.compression()}")
                print(f"Server hostname: {self.sock.server_hostname}")
                print(f"Socket timeout: {self.sock.gettimeout()}")

        except Exception as e:
            print(f"Failed to connect: {e}")
            self.sock = None

    def send_file(self, file_path):
        if not self.sock:
            print("No connection established.")
            return

        try:
            with open(file_path, 'rb') as file:
                data = file.read()
                data_size = len(data)

                # Send file size header
                size_header = data_size.to_bytes(8, byteorder='big')
                self.sock.sendall(size_header)

                # Measure only the data transfer time (not ACK reception
                    )
                start_time = time.time()
                self.sock.sendall(data)
                end_time = time.time()

                # Wait for acknowledgment (but don't include in timing)
                ack = self.sock.recv(1024)

                if ack:
                    ack_decoded = ack.decode('utf-8')
                    print(f"Server acknowledged: {ack_decoded}")

                duration = end_time - start_time
                average_speed = data_size / duration if duration > 0
                    else 0

                # Update stats
                self.stats['data_size'] = data_size
                self.stats['transfer_time'] = duration
                self.stats['average_speed'] = average_speed
                self.stats['timestamp'] = datetime.now().strftime("%Y-%m
                    -%d %H:%M:%S")

```

```
                print(f"Sent {data_size} bytes in {duration:.6f} seconds
                    . Average speed: {average_speed:.2f} bytes/second.")

                # Log performance
                log_performance(data_size, duration, self.use_tls)

        except IOError as e:
            print(f"Failed to read/send file: {e}")
        finally:
            self.sock.close()
```

5 Resultados e Análise

O uso do TLS, apesar de essencial para assegurar confidencialidade e integridade, acarreta um custo adicional de desempenho, conhecido como *overhead*. Esse custo resulta diretamente das operações criptográficas realizadas durante a negociação da conexão e na proteção dos dados transmitidos.

Assim, o experimento foi conduzido para comparar o desempenho da transferência de dados usando TCP puro e TLS. Para isso, foi utilizado um arquivo pequeno (`test_file.txt`, com 195 bytes) e realizado um conjunto de **10 execuções** para cada protocolo. Em cada rodada, o script `run_performance_tests.py` estabeleceu a conexão, transferiu o arquivo e registrou automaticamente o tempo total e o *throughput*.

Os resultados finais apresentados são as médias das 10 repetições por protocolo, o que reduz variações pontuais e permite uma análise mais confiável do impacto do TLS sobre o desempenho.

Métrica	TCP Puro	TLS	Impacto
Tempo Médio	0.0077s	0.0123s	**+58.62%**
Throughput Médio	43.41 MB/s	17.58 MB/s	**-59.50%**

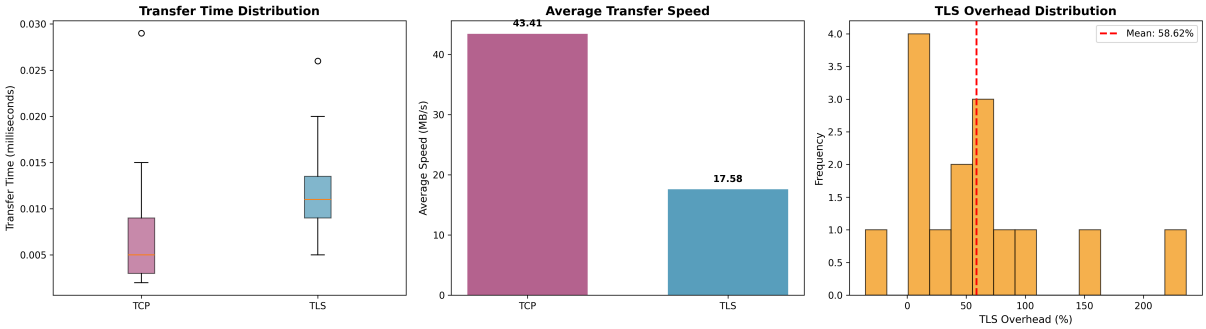

Figura 5: Gráfico de comparação TCP Puro x TLS

5.1 Interpretação do Overhead

Os resultados mostram que o uso de TLS aumentou o tempo médio de transferência de 0.0077s para 0.0123s, o que representa um acréscimo de 58.62%. Esse aumento corresponde a um **Fator de Desaceleração de aproximadamente 1.60x**, indicando que a transmissão protegida levou cerca de 60% mais tempo que o TCP puro.

Os motivos principais para essa desaceleração são:

- **Handshake Inicial**: Antes de enviar qualquer dado, o TLS precisa negociar algoritmos, validar certificado e realizar a troca de chaves (ECDHE), introduzindo latência adicional.
- **Criptografia e Descriptografia**: Cada bloco de dados precisa ser cifrado pelo cliente e decifrado pelo servidor. Esse processamento contínuo, embora rápido em arquivos pequenos, ainda adiciona custo de CPU perceptível.
- **Sobrecarga de Protocolo**: Cada registro TLS inclui metadados e um *authentication tag*, aumentando levemente o tamanho transmitido e contribuindo para a queda de vazão.

Esses fatores explicam a redução de 59.50% no throughput, que caiu de 43.41 MB/s (TCP) para 17.58 MB/s (TLS).

6 Conclusão

Os experimentos demonstram que o uso de TLS introduz um custo de desempenho mensurável, aumentando o tempo de transferência em cerca de 1.60x e reduzindo o throughput em aproximadamente 60%. Apesar desse impacto, o TLS permanece indispensável, pois garante confidencialidade, integridade e autenticação — requisitos fundamentais em qualquer comunicação moderna.

Assim, o relatório confirma que existe um trade-off claro entre segurança e desempenho: o TLS adiciona latência e processamento, mas esses custos são plenamente justificáveis frente aos benefícios de proteção dos dados.