FACULDADE CESAR SCHOOL
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DISCIPLINA DE INFRAESTRUTURA DE COMUNICAÇÃO
RELATÓRIO 2ª UNIDADE

HENRIQUE MAGALHÃES DE LUCENA
IGOR FELIPE WANDERLEY DA SILVA
JOÃO VICTOR FERRAZ GONÇALVES
MARIA JÚLIA DE OLIVEIRA TELES DE MENEZES
MARIA LUÍSA COIMBRA LIMA
PEDRO WANDERLEY DE LIRA SANTOS

RECIFE
NOVEMBRO/2024

HENRIQUE MAGALHÃES DE LUCENA
IGOR FELIPE WANDERLEY DA SILVA
JOÃO VICTOR FERRAZ GONÇALVES
MARIA JÚLIA DE OLIVEIRA TELES DE MENEZES
MARIA LUÍSA COIMBRA LIMA
PEDRO WANDERLEY DE LIRA SANTOS

RELATÓRIO 2ª UNIDADE

Relatório apresentado para obtenção de nota parcial da 2ª unidade da disciplina de Infraestrutura de Comunicação do Curso de Graduação em Ciência da Computação da Faculdade Cesar School, sob orientação do Prof. Petrônio Gomes Lopes Junior.

RECIFE
NOVEMBRO/2024

1. INTRODUÇÃO

1.1 Objetivo do Projeto

O presente trabalho teve como objetivo desenvolver uma aplicação cliente-servidor que, na camada de aplicação, fosse capaz de fornecer transporte confiável de dados em um cenário com perdas e erros simulados. Este desafio exigiu a implementação de mecanismos robustos de controle de fluxo e controle de congestionamento, seguindo princípios fundamentais da comunicação em redes.

A aplicação foi estruturada utilizando sockets para estabelecer a comunicação entre cliente e servidor. Adicionalmente, foi proposto e descrito um protocolo de aplicação que definiu as regras para as interações entre as partes, como requisições e respostas.

Entre as funcionalidades essenciais estão a simulação de falhas de integridade e perdas de mensagens, a verificação e tratamento de pacotes perdidos, e a confirmação de entrega por meio de métodos individuais ou em grupo. Também foram implementados mecanismos como soma de verificação, temporizadores, números de sequência, reconhecimentos positivos e negativos, e o controle de janela deslizante para garantir o paralelismo e a eficiência da transmissão.

Além disso, o projeto ofereceu uma oportunidade de explorar métodos avançados, como a negociação entre cliente e servidor para utilização dos protocolos de repetição seletiva ou Go-Back-N, bem como a atualização dinâmica da janela de recepção e da janela de congestionamento.

Com essa abordagem, o trabalho visou consolidar os conhecimentos adquiridos sobre redes de computadores, destacando a importância de garantir a confiabilidade no transporte de dados em cenários reais e simulados.

2. Arquitetura da Solução

2.1 Visão Geral da Arquitetura Cliente-Servidor

A aplicação cliente-servidor foi desenvolvida utilizando a biblioteca **socket**, que permite comunicação entre dispositivos conectados em rede. A interação segue o modelo tradicional cliente-servidor, onde:

- **Servidor**: Aguarda conexões e processa pacotes recebidos, garantindo a ordem de entrega, reconhecimentos, e controle de fluxo.
- **Cliente**: Envia pacotes ao servidor com mecanismos para simular erros de integridade e gerenciar o envio baseado em controle de congestionamento.

A comunicação entre cliente e servidor é realizada via **sockets TCP**, permitindo troca confiável de dados. O servidor é configurado para escutar conexões em uma porta específica (63214), enquanto o cliente inicia a conexão.

Fluxo de Operação:

1. O cliente realiza um *handshake* com o servidor para negociar parâmetros como tamanho da janela e protocolo (Selective Repeat ou Go-Back-N).
2. O cliente envia pacotes contendo sequência, dados e soma de verificação.
3. O servidor valida pacotes, envia reconhecimentos (ACK ou NAK), e gerencia a janela de recepção.
4. O controle de fluxo e congestionamento é gerenciado dinamicamente com base nas confirmações e perdas detectadas.

Exemplo de Código do Servidor (Criação do Socket e Aceitação de Conexões):

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind((self.server_host, self.server_port))

server_socket.listen(5)

print("Server is running and waiting for connections...")

client_socket, client_address = server_socket.accept()

print(f"Connection established with {client_address}.")
```

Exemplo de Código do Cliente (Inicialização do Socket e Conexão):

```
self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

self.client_socket.connect((self.server_host, self.server_port))
```

2.2 Protocolo de Aplicação

O protocolo desenvolvido estabelece as regras de comunicação entre cliente e servidor. Ele define o formato das mensagens, os tipos de requisições e respostas, e as ações a serem tomadas com base nas mensagens trocadas.

Formato das Mensagens: As mensagens são estruturadas no formato TIPO|SEQUÊNCIA|DADOS|CHECKSUM. Os campos têm os seguintes significados:

- **TIPO**: Identifica o tipo da mensagem (e.g., SEND, ACK, NAK).
- **SEQUÊNCIA**: Número único associado a cada pacote.
- **DADOS**: Conteúdo do pacote.
- **CHECKSUM**: Soma de verificação para garantir a integridade.

Regras de Requisição e Resposta:

- **Handshake**: O cliente inicia a conexão com uma mensagem HANDSHAKE para negociar protocolo e janela.

  Cliente: `HANDSHAKE|PROTOCOL|<protocolo>|WINDOW|<tamanho_da_janela>`

  Servidor: `ACK_HANDSHAKE|PROTOCOL|<protocolo>|WINDOW|<tamanho_da_janela>`

- **Envio de Dados**: O cliente envia pacotes com o tipo SEND. O servidor valida e responde com:
  - **ACK**: Quando o pacote é recebido corretamente.
  - **NAK**: Quando há erro de integridade ou pacotes fora de ordem.
- **Controle de Fluxo**: Ambas as partes atualizam dinamicamente as janelas de transmissão e recepção com base nos pacotes trocados.

Exemplo de Handshake (Cliente):

```
handshake_message=f"HANDSHAKE|PROTOCOL|{self.protocol_type}|WINDOW|{self.max_w
indow}"

self.client_socket.sendall(f"{handshake_message}\n".encode())
```

```
print(f"Sent: {handshake_message}")

server_response = self.client_socket.recv(1024).decode().strip()

print(f"Handshake confirmed by server: {server_response}")
```

Exemplo de Processamento de Mensagem (Servidor):

```
def process_message(self, connection, message):

    message_parts = message.split("|")

    if message_parts[0] == "SEND":

         self.handle_send(connection, int(message_parts[1]), message_parts[2],
int(message_parts[3]))

    elif message_parts[0] == "NAK":

        self.send_message(connection, "NAK", sequence_number)
```

Com esta arquitetura e protocolo, o sistema garante transporte confiável, mesmo em um ambiente com simulação de falhas.

3. Implementação do Cliente

3.1 Envio de Pacotes

O cliente foi projetado para enviar pacotes individualmente ou em rajadas, conforme configurado pelo usuário. Cada pacote contém informações como número de sequência, dados e soma de verificação (checksum). O envio respeita a janela de congestionamento, limitando o número de pacotes que podem ser enviados simultaneamente.

Fluxo do Envio:

1. O cliente carrega os dados a serem enviados (mensagens do arquivo `bandas_forro.txt`).
2. Cada pacote é formatado no padrão `SEND|SEQUÊNCIA|DADOS|CHECKSUM.`
3. Os pacotes são enviados respeitando a janela de congestionamento (`cwnd`), que cresce ou diminui conforme os ACKs e NAKs recebidos.

Exemplo de Envio de Pacote:

```
def send_message_packet(self, sequence_number):

    message_content = self.data_buffer[sequence_number - 1]

    checksum = self.calculate_checksum(message_content)

    message_content = f"SEND|{sequence_number}|{message_content}|{checksum}"

    self.client_socket.sendall(f"{message_content}\n".encode())

        print(f"[Packet Sent] Packet {sequence_number} sent with checksum {checksum}.")
```

O cliente também permite enviar pacotes em rajadas, utilizando uma fila (`deque`) para gerenciar pacotes pendentes, garantindo envio em ordem até que a janela permita novos envios.

3.2 Simulação de Erros

Para simular falhas de integridade, o cliente permite alterar intencionalmente o checksum de pacotes específicos, criando erros que serão detectados pelo servidor. O usuário define os números dos pacotes que devem conter erros durante a configuração inicial.

Introdução de Erros:

1. Durante o envio, o cliente verifica se o número do pacote está na lista de pacotes com erros (`packets_with_error`).
2. Caso afirmativo, o checksum é alterado para um valor incorreto.

Exemplo de Simulação de Erro:

```
if sequence_number in self.packets_with_error:

    checksum = (checksum + 1) & 0xFFFF

     print(f"[Checksum Injection] Intentionally altering checksum for Packet {sequence_number}. New checksum: {checksum}")
```

Essa funcionalidade permite testar a capacidade do servidor de detectar pacotes corrompidos e reagir adequadamente com NAKs.

3.3 Controle de Janela

O cliente respeita a janela de recepção do servidor, garantindo que os pacotes enviados estejam dentro do intervalo permitido. A janela é ajustada dinamicamente com base nos ACKs recebidos e nos NAKs que solicitam retransmissão.

Dinâmica da Janela:

- O cliente mantém uma fila de pacotes pendentes (`pending_packets`) e só envia novos pacotes se estiverem dentro da janela.
- A cada ACK recebido, o cliente avança na fila e envia novos pacotes.

Exemplo de Atualização da Janela:

```
def send_pending_packets(self):

    while self.pending_packets and len(self.sent_packets) < self.cwnd:

        next_packet = self.pending_packets.popleft()

        if next_packet not in self.acknowledged_packets:

            self.send_message_packet(next_packet)
```

Essa abordagem evita o envio excessivo de pacotes, alinhando o envio à capacidade do servidor de processá-los.

3.4 Controle de Congestionamento

O controle de congestionamento é implementado com uma abordagem dinâmica, ajustando a janela de congestionamento (`cwnd`) conforme os eventos observados:

- **Crescimento Exponencial**: Durante o estágio de *slow start*, a `cwnd` aumenta exponencialmente a cada ACK recebido, até atingir o limiar de congestionamento (`ssthresh`).
- **Crescimento Linear**: Após atingir o `ssthresh`, a `cwnd` cresce linearmente para evitar sobrecarga na rede.
- **Redução em Perdas**: Ao receber um NAK ou detectar uma perda, o `ssthresh` é reduzido para metade da `cwnd`, e a `cwnd` reinicia em 1.

Exemplo de Controle de Congestionamento:

```
if response_type == "ACK":

    if self.cwnd < self.ssthresh:

        self.cwnd *= 2 # Crescimento exponencial

    else:

        self.cwnd += 1  # Crescimento linear

elif response_type == "NAK":

    self.ssthresh = max(1, self.cwnd // 2)  # Reduzir threshold

    self.cwnd = 1  # Reiniciar janela
```

Essa lógica garante que o cliente reaja às condições da rede, equilibrando eficiência e confiabilidade.

4. Implementação do Servidor

4.1 Recepção de Pacotes

O servidor foi projetado para processar pacotes enviados pelo cliente, verificar sua integridade e responder com confirmações (ACK) ou negativas (NAK). Ao receber um pacote, o servidor:

1. Extrai e valida o número de sequência, o conteúdo, e o checksum.
2. Decide o tipo de resposta com base no status do pacote:
    - **ACK**: Pacote recebido corretamente e em ordem.
    - **NAK**: Pacote fora de ordem ou com falha de integridade.

Exemplo de Recepção e Resposta:

```
def process_message(self, connection, message):

    message_parts = message.split("|")

    sequence_number = int(message_parts[1])

    content = message_parts[2]

    checksum_received = int(message_parts[3])

    calculated_checksum = sum(ord(c) for c in content) & 0xFFFF



    if checksum_received != calculated_checksum:

        print(f"[Checksum Error] Packet {sequence_number} invalid. Sending NAK.")

        self.send_message(connection, "NAK", sequence_number)

    else:

        self.handle_send(connection, sequence_number, content, checksum_received)
```

Essa abordagem garante que o servidor identifique e trate pacotes inválidos ou em ordem incorreta.

4.2 Simulação de Falhas

O servidor pode simular falhas ao marcar pacotes que não serão confirmados e ao introduzir erros nas confirmações enviadas ao cliente. Isso permite testar a capacidade do cliente de reagir a falhas.

- **Pacotes Não Confirmados**: O servidor pode decidir não enviar um ACK para determinados pacotes, simulando perdas.
- **Erros nas Confirmações**: O servidor altera intencionalmente o conteúdo de confirmações para introduzir falhas.

Exemplo de Falhas Simuladas:

```
def send_message(self, connection, response_type, sequence_number):

    if response_type == "ACK" and sequence_number in self.failed_ack_packets:

        print(f"[Simulated Error] Not sending ACK for Packet {sequence_number}.")

        return

    elif response_type == "ACK" and sequence_number in self.corrupt_ack_packets:

        response_type = "ERR"

                    print(f"[Simulated Error] Sending corrupted ACK for Packet
{sequence_number}.")

    connection.sendall(f"{response_type}|{sequence_number}|CONFIRM\n".encode())
```

4.3 Confirmações Negativas

O servidor envia confirmações negativas (NAKs) quando detecta problemas nos pacotes recebidos, como:

1. Checksum inválido.
2. Pacotes fora da janela de recepção.
3. Pacotes duplicados.

O NAK informa ao cliente que o pacote precisa ser retransmitido.

Exemplo de Envio de NAK:

```
def process_invalid_packet(self, connection, sequence_number, content):

    if sequence_number not in self.current_receive_window:

            print(f"[Out-of-Bounds Packet] Packet {sequence_number} rejected.
Sending NAK.")

        self.send_message(connection, "NAK", sequence_number)
```

Esse mecanismo permite que o cliente retransmita pacotes problemáticos, mantendo a integridade dos dados.

4.4 Negociação de Método de Retransmissão

Durante o handshake inicial, o cliente e o servidor negociam o método de retransmissão a ser usado: **Selective Repeat (SR)** ou **Go-Back-N (GBN)**. O servidor valida a escolha do protocolo e configura o comportamento para lidar com retransmissões de acordo.

Fluxo de Negociação:

1. O cliente envia uma mensagem de handshake indicando o protocolo desejado.
2. O servidor valida e confirma o protocolo se estiver disponível.

Exemplo de Negociação de Protocolo:

```
def perform_handshake(self, connection):

    handshake_data = connection.recv(1024).decode().strip()

    _, _, protocol, _, window_size = handshake_data.split("|")

    if protocol == self.protocol_type:

        response = f"ACK_HANDSHAKE|PROTOCOL|{protocol}|WINDOW|{self.receive_window_size}"

        connection.sendall(f"{response}\n".encode())

        print(f"Handshake successful. Protocol: {protocol}, Window Size: {window_size}.")

    else:

        print("[Handshake Error] Protocol mismatch.")

        connection.close()
```

Essa negociação permite que ambos os lados estejam sincronizados quanto ao comportamento de retransmissão.

4.5 Atualização da Janela de Recepção

O servidor mantém uma janela de recepção que define o intervalo de números de sequência aceitos. Essa janela é informada ao cliente durante o handshake e atualizada dinamicamente à medida que pacotes são recebidos e processados.

Funcionamento:

1. A janela inicial é definida com base no número esperado (`expected_sequence`) e no tamanho da janela (`receive_window_size`).
2. Ao processar um pacote em ordem, o servidor avança a janela.
3. Pacotes fora da janela são descartados ou armazenados temporariamente para processamento futuro.

Exemplo de Atualização da Janela:

```
def update_receive_window(self):

    self.expected_sequence += 1

    self.current_receive_window = list(

    range(self.expected_sequence, self.expected_sequence + self.receive_window_size)

    )

    print(f"[Flow Control] Updated receive window: {self.current_receive_window}")
```

Essa lógica assegura que o servidor aceite pacotes válidos e em ordem, enquanto sinaliza ao cliente quando retransmissões são necessárias.

5. Características do Transporte Confiável de Dados

5.1 Soma de Verificação

A soma de verificação (**checksum**) é utilizada para garantir a integridade dos pacotes transmitidos entre o cliente e o servidor. Esse mecanismo permite detectar alterações nos dados durante a transmissão.

Implementação:

1. Antes de enviar um pacote, o cliente calcula a soma de verificação somando os valores ASCII de cada caractere da mensagem e aplicando uma operação AND com `0xFFFF` (para limitar o valor a 16 bits).
2. O checksum calculado é incluído no pacote enviado.
3. No servidor, o checksum é recalculado e comparado ao valor recebido. Em caso de divergência, o pacote é considerado inválido.

Exemplo de Código (Cálculo do Checksum no Cliente):

```
def calculate_checksum(self, message):

    checksum = sum(ord(c) for c in message) & 0xFFFF

          print(f"[Checksum  Calculation]  Message:  '{message}',  Checksum:
{checksum}")

    return checksum
```

Exemplo de Validação no Servidor:

```
calculated_checksum = sum(ord(c) for c in content) & 0xFFFF

if checksum_received != calculated_checksum:

     print(f"[Checksum Error] Packet {sequence_number} has invalid checksum.
Sending NAK.")

    self.send_message(connection, "NAK", sequence_number)
```

5.2 Temporizador

O temporizador é utilizado pelo cliente para gerenciar o tempo de espera por uma confirmação (ACK ou NAK) do servidor. Se uma resposta não for recebida dentro do tempo limite, o pacote é retransmitido.

Funcionamento:

1. Ao enviar um pacote, o cliente inicia um temporizador associado ao número de sequência.
2. Se o tempo limite expirar sem o recebimento de uma resposta, o cliente retransmite o pacote.
3. O temporizador é cancelado quando um ACK é recebido.

Exemplo de Temporizador no Cliente:

```
self.message_timeout = 2  # Tempo limite em segundos

if sequence_number not in self.acknowledged_packets:

    print(f"[Timeout] Packet {sequence_number} not acknowledged. Retrying...")

    self.send_message_packet(sequence_number)
```

5.3 Número de Sequência e Reconhecimento

Os números de sequência são utilizados para identificar unicamente cada pacote transmitido. Esses números permitem ao servidor:

- Verificar a ordem dos pacotes.
- Detectar pacotes duplicados ou fora de ordem.
- Reconhecer pacotes recebidos corretamente com **ACKs** ou sinalizar problemas com **NAKs**.

Tratamento no Cliente:

- O cliente envia pacotes com números de sequência incrementais.
- Pacotes duplicados ou não reconhecidos são retransmitidos.

Tratamento no Servidor:

- O servidor mantém um número esperado de sequência (`expected sequence`).
- Pacotes fora da sequência são armazenados em buffer ou descartados, dependendo do protocolo.

Exemplo de Envio e Reconhecimento (Servidor):

```
if sequence_number == self.expected_sequence:

    self.process_in_order_packet(connection, sequence_number, content)

    self.send_message(connection, "ACK", sequence_number)

else:
```

```
    self.process_invalid_packet(connection, sequence_number, content)
```

5.4 Janela e Paralelismo

A janela deslizante é uma técnica que permite o envio de múltiplos pacotes antes de receber confirmações, garantindo maior eficiência ao utilizar o canal de comunicação.

Funcionamento:

1. O cliente mantém uma janela de congestionamento (`cwnd`) que define quantos pacotes podem ser enviados simultaneamente.
2. A janela avança conforme os ACKs são recebidos.
3. O servidor mantém uma janela de recepção que determina quais pacotes podem ser aceitos.

Interação com Paralelismo:

- No cliente, múltiplos pacotes podem ser enviados enquanto espera-se os ACKs.
- No servidor, pacotes fora de ordem são armazenados em buffer até que os pacotes anteriores sejam processados.

Exemplo de Janela no Cliente:

```
while self.pending_packets and len(self.sent_packets) < self.cwnd:

    next_packet = self.pending_packets.popleft()

    if next_packet not in self.acknowledged_packets:

        self.send_message_packet(next_packet)
```

Exemplo de Janela no Servidor:

```
def update_receive_window(self):

    self.expected_sequence += 1

    self.current_receive_window = list(

                    range(self.expected_sequence,  self.expected_sequence  +
self.receive_window_size)

    )

                print(f"[Flow    Control]    Updated    receive    window:
{self.current_receive_window}")
```

Limitação do Tamanho das Mensagens

Para garantir a consistência da comunicação e evitar problemas causados pelo envio de mensagens excessivamente grandes, foi implementado um limite máximo de tamanho de 256 bytes para cada mensagem. Essa restrição foi configurada tanto no cliente quanto no servidor.

No Cliente: Antes de enviar uma mensagem, o cliente verifica seu tamanho em bytes. Caso o tamanho exceda o limite definido, a mensagem é truncada para atender à restrição, e um aviso é exibido no terminal.

```
if len(message_content.encode()) > MAX_MESSAGE_SIZE:

    print(Fore.RED + f"[Warning] Message for Packet {sequence_number} exceeds
{MAX_MESSAGE_SIZE} bytes. Truncating.")

    message_content =
message_content.encode()[:MAX_MESSAGE_SIZE].decode(errors='ignore')
```

No Servidor: Ao receber um pacote, o servidor verifica o tamanho da mensagem contida. Se a mensagem exceder o limite, ela também é truncada, e um aviso é exibido no terminal.

```
if len(content.encode()) > MAX_MESSAGE_SIZE:

      print(Fore.RED + f"[Warning] Packet {sequence_number} content exceeds
{MAX_MESSAGE_SIZE} bytes. Truncating.")

    content = content.encode()[:MAX_MESSAGE_SIZE].decode(errors='ignore')
```

Durante o truncamento:

1. Os primeiros **256 bytes** da mensagem são mantidos.
2. Os **bytes excedentes** são descartados.
3. A mensagem resultante ainda é enviada e processada, mas em um formato reduzido.

O código do cliente e servidor garante que:

- Nenhuma mensagem será completamente rejeitada ou perdida devido ao tamanho.
- Apenas a parte que excede o limite será descartada (truncada).

Essa funcionalidade evita problemas como:

- Sobrecarga de buffers na aplicação ou no protocolo subjacente.
- Erros de comunicação causados por tamanhos excessivos de pacotes.

- Possíveis incompatibilidades com limites de tamanho impostos por outros sistemas ou protocolos.

A limitação do tamanho das mensagens reforça a robustez da aplicação, garantindo que apenas pacotes dentro do limite permitido sejam processados. Além disso, ela aumenta a compatibilidade com padrões comuns de redes e sistemas distribuídos.

6. Simulação de Falhas e Erros

6.1 Simulação de Falhas de Integridade

Falhas de integridade são introduzidas no sistema para testar a capacidade do servidor de identificar pacotes corrompidos e reagir de maneira apropriada. Essa simulação é feita pelo cliente, alterando intencionalmente o checksum de pacotes específicos antes do envio.

Funcionamento:

1. O cliente recebe, no início da execução, uma lista de pacotes que devem conter falhas de integridade.
2. Durante o envio de pacotes, o checksum desses pacotes é modificado.
3. O servidor verifica o checksum ao receber o pacote. Caso o valor recebido seja diferente do esperado, o servidor envia um **NAK** para solicitar a retransmissão.

Exemplo de Simulação no Cliente:

```
if sequence_number in self.packets_with_error:

    checksum = (checksum + 1) & 0xFFFF  # Modificar checksum para simular erro

     print(f"[Checksum Injection] Intentionally altering checksum for Packet {sequence_number}. New checksum: {checksum}")
```

Exemplo de Detecção no Servidor:

```
calculated_checksum = sum(ord(c) for c in content) & 0xFFFF

if checksum_received != calculated_checksum:

    print(f"[Checksum Error] Packet {sequence_number} invalid. Sending NAK.")

    self.send_message(connection, "NAK", sequence_number)
```

Essa abordagem permite validar a robustez do sistema contra pacotes corrompidos e garante que a comunicação se mantenha consistente.

6.2 Perda de Pacotes

A perda de pacotes é simulada tanto no cliente quanto no servidor para testar a resiliência do sistema ao lidar com dados ausentes.

No Cliente:

- O cliente mantém um contador global para monitorar o número de pacotes enviados.
- Determinados pacotes podem ser omitidos intencionalmente no envio, simulando sua perda.

Exemplo de Simulação de Perda no Cliente:

```
if sequence_number in self.lost_packets:

       print(f"[Simulated Loss] Packet {sequence_number} intentionally not sent.")

    return False
```

No Servidor:

- O servidor pode ser configurado para ignorar confirmações de determinados pacotes, simulando a perda desses dados na rede.

Exemplo de Simulação de Perda no Servidor:

```
if sequence_number in self.failed_ack_packets:

    print(f"[Simulated Error] Not sending ACK for Packet {sequence_number}.")

    return
```

Impacto na Comunicação Cliente-Servidor:

- Quando um pacote é perdido, o cliente não recebe a confirmação correspondente dentro do tempo limite.
- O temporizador do cliente aciona a retransmissão do pacote, permitindo que o servidor receba o dado ausente.

Exemplo de Retransmissão no Cliente:

```
if sequence_number not in self.acknowledged_packets:

    print(f"[Timeout] Packet {sequence_number} not acknowledged. Retrying...")

    self.send_message_packet(sequence_number)
```

7. Controle de Fluxo e Congestionamento

7.1 Mecanismos de Controle de Fluxo

O controle de fluxo é implementado para garantir que o servidor processe os pacotes na velocidade adequada, evitando sobrecarga e perda de dados. Para isso, o servidor mantém uma **janela de recepção dinâmica** que informa ao cliente o intervalo de pacotes que podem ser aceitos.

Funcionamento:

1. Durante o handshake, o servidor e o cliente negociam o tamanho inicial da janela de recepção.
2. A cada pacote recebido em ordem, o servidor avança a janela e envia um **ACK** ao cliente.
3. Se pacotes fora da ordem são recebidos, eles são armazenados em buffer até que o pacote esperado seja processado.

Exemplo de Atualização da Janela no Servidor:

```
def update_receive_window(self):

    self.expected_sequence += 1

    self.current_receive_window = list(

                      range(self.expected_sequence,  self.expected_sequence  +
self.receive_window_size)

    )

                  print(f"[Flow    Control]    Updated    receive    window:
{self.current_receive_window}")
```

Efeito no Cliente:

- O cliente respeita a janela informada pelo servidor e só envia pacotes dentro do intervalo permitido.
- A coordenação entre as janelas de recepção e envio garante que o servidor não seja sobrecarregado com pacotes.

Exemplo no Cliente:

```
if sequence_number > self.cwnd:

       print(f"[Flow  Control]  Packet  {sequence_number}  waiting  for  window
availability.")

    return False
```

Esses mecanismos asseguram a entrega eficiente dos pacotes e previnem a perda de dados devido a congestionamentos no servidor.

7.2 Controle de Congestionamento

O controle de congestionamento é implementado no cliente, com uma janela de congestionamento dinâmica (`cwnd`) que ajusta o número de pacotes enviados com base no estado da rede.

Mecanismos Implementados:

1. **Slow Start**:
   - A janela de congestionamento começa pequena (`cwnd = 1`).
   - Durante a fase de *slow start*, a janela cresce exponencialmente a cada **ACK** recebido, dobrando de tamanho até atingir o limiar de congestionamento (`ssthresh`).
2. **Congestion Avoidance**:
   - Após atingir o `ssthresh`, o crescimento da janela torna-se linear, aumentando um pacote por ciclo de confirmação.
3. **Detecção de Perdas**:
   - Quando um **NAK** ou perda de pacote é detectado, o `ssthresh` é reduzido pela metade, e o `cwnd` é reiniciado para 1.
4. **Retransmissão Rápida**:
   - Caso pacotes sejam perdidos ou confirmações duplicadas sejam recebidas, o cliente retransmite apenas os pacotes afetados.

Exemplo de Crescimento da Janela (Slow Start e Avoidance):

```
if response_type == "ACK":

    if self.cwnd < self.ssthresh:

        self.cwnd += 1  # Crescimento exponencial

              print(f"[Congestion  Control]  Slow  Start:  cwnd  increased  to
{self.cwnd}.")

    else:

        self.cwnd += 1  # Crescimento linear

         print(f"[Congestion Control] Congestion Avoidance: cwnd increased to
{self.cwnd}.")
```

Exemplo de Redução de Janela após Perda:

```
if response_type == "NAK":
```

```
    self.ssthresh = max(1, self.cwnd // 2)  # Reduzir limiar

    self.cwnd = 1  # Reiniciar janela

        print(f"[Congestion Control] Packet loss detected. cwnd reset to
{self.cwnd}, ssthresh set to {self.ssthresh}.")
```

Esses mecanismos permitem que o cliente se adapte dinamicamente às condições da rede, prevenindo sobrecarga e otimizando o desempenho.

8. Protocolo de Aplicação

8.1 Estrutura das Mensagens

Mensagens do Cliente:
Cada mensagem enviada pelo cliente é estruturada da seguinte forma:

- **HEADER**: Um identificador do tipo de mensagem (por exemplo, DATA).
- **SEQUENCE_NUMBER**: Número de sequência do pacote enviado.
- **CHECKSUM**: Hash (pode ser MD5 ou SHA256) para verificar a integridade do pacote.
- **PAYLOAD**: Dados reais da mensagem (por exemplo, o conteúdo da mensagem).

Mensagens do Servidor:
As respostas enviadas pelo servidor seguem o seguinte formato:

- **HEADER**: Identificador do tipo de resposta (por exemplo, ACK ou NACK).
- **SEQUENCE_NUMBER**: Número de sequência do pacote confirmado ou rejeitado.
- **STATUS**: Status da resposta, que pode ser OK ou ERROR.

8.2 Tipos de Mensagens

Mensagens do Cliente:

1. **DATA**: Envia um pacote de dados para o servidor.
2. **BATCH**: Envia vários pacotes em sequência.
3. **ERROR**: Solicita a simulação de erro em pacotes específicos.

Mensagens do Servidor:

1. **ACK**: Confirma o recebimento de um pacote.
2. **NACK**: Solicita que o cliente retransmita o pacote devido a erro ou falta.
3. **STATUS**: Informa ao cliente sobre o estado do servidor ou da comunicação.

8.3 Regras do Protocolo

8.3.1 Estabelecimento de Conexão:

O cliente inicia a comunicação enviando a mensagem `HELLO`:

O servidor responde com: `WELCOME|SERVER|[Protocolo: GBN/SR]`

8.3.2 Envio de Pacotes:

- O cliente envia pacotes do tipo DATA ou BATCH.
- O servidor responde com:
    - `ACK|SEQUENCE_NUMBER|OK`: Caso o pacote seja recebido corretamente.
    - `NACK|SEQUENCE_NUMBER|ERROR`: Caso haja erro na mensagem (por exemplo, erro de integridade).

8.3.3 Retransmissão:

- Quando o cliente não recebe um ACK dentro do tempo estipulado, ele retransmite os pacotes que não foram confirmados. Isso ocorre automaticamente em caso de timeout.

8.3.4 Controle de Fluxo:

- O servidor informa ao cliente a janela de recepção disponível por meio de mensagens de **STATUS**. O cliente ajusta a quantidade de dados que pode enviar com base na janela de recepção informada.

8.3.5 Controle de Congestionamento:

- O cliente ajusta dinamicamente a janela de envio com base na quantidade de pacotes perdidos e nas confirmações duplicadas que recebe.

8.3.6 Simulação de Erros:

O cliente pode simular erros em pacotes específicos ao enviar a mensagem: `ERROR|SEQUENCE_NUMBER`

- O servidor pode responder com um **NACK** se detectar um erro na integridade dos dados.

8.3.7 Negociação de Protocolo:

- Durante o estabelecimento da conexão, o cliente e o servidor negociam qual protocolo de retransmissão será utilizado:
    - **Go-Back-N (GBN)**: O cliente retransmite pacotes de forma contínua até que receba a confirmação do pacote anterior.
    - **Selective Repeat (SR)**: O cliente retransmite apenas pacotes específicos que não foram confirmados.

9. Manual de Utilização

9.1 Instruções para o Cliente e Servidor

Requisitos de Instalação e Configuração:

1. **Sistema Operacional**: O código é compatível com sistemas baseados em Windows, macOS ou Linux.
2. **Python**: Certifique-se de ter o **Python 3.8** ou superior instalado.
3. **Dependências**:
    - Instale a biblioteca `colorama` para a formatação das mensagens no terminal.

Execute o comando abaixo para instalar as dependências:
`pip install colorama`

Passo a Passo para Rodar o Servidor:

1. Navegue até o diretório onde o arquivo `server.py` está localizado.

Execute o script utilizando o comando:

`python server.py`

2. Insira os parâmetros iniciais, como:
    - O protocolo a ser utilizado: `SR` (Selective Repeat) ou `GBN` (Go-Back-N).
    - Tamanho da janela de recepção.
    - Ativação ou não de confirmações cumulativas.

Passo a Passo para Rodar o Cliente:

1. Navegue até o diretório onde o arquivo `client.py` está localizado.

Execute o script utilizando o comando:

`python client.py`

2. Insira os parâmetros iniciais, como:
    - Tamanho inicial da janela.
    - Total de pacotes a serem enviados.
    - Protocolo a ser utilizado (`SR` ou `GBN`).
    - Lista de pacotes com erros, informada como números separados por vírgulas (e.g., `1,3,5`).

9.2 Comandos e Fluxos

Fluxo Geral:

1. O **servidor** deve ser iniciado primeiro e permanecer aguardando conexões.
2. O **cliente** é iniciado posteriormente e estabelece a conexão com o servidor através do processo de handshake.

Comandos no Cliente:

- **Enviar Pacotes**: O cliente envia pacotes automaticamente após o início do programa, respeitando as configurações de janela e protocolo.
- **Configurar Falhas**: Ao iniciar o cliente, é possível simular falhas configurando:
    - Pacotes com falhas de integridade (exemplo: `2,4` para introduzir erros nos pacotes 2 e 4).
    - Total de mensagens a serem enviadas.

Exemplo de Interação no Cliente:

```
Enter initial window size: 5

Enter total number of messages to send: 10

Choose protocol (SR for Selective Repeat, GBN for Go-Back-N): SR

Enter packet numbers to simulate errors (comma-separated): 3,6
```

Visualização dos Resultados:

- No **cliente**, as mensagens exibem:
    - Pacotes enviados, incluindo número de sequência e checksum.
    - Respostas recebidas do servidor (ACKs, NAKs).
    - Mudanças na janela de congestionamento.

Exemplo de Saída no Cliente:

```
[Packet Sent] Packet 3 sent with checksum 12345.

[NAK Received] NAK for Packet 3 received. Retransmitting...

[Congestion Control] cwnd reset to 1, ssthresh set to 3.
```

- No **servidor**, as mensagens exibem:
    - Pacotes recebidos, com validação de checksum.
    - Atualizações da janela de recepção.
    - Respostas enviadas ao cliente (ACKs, NAKs).

Exemplo de Saída no Servidor:

```
[Message Received] Packet 3: Type=SEND, Content='example', Checksum=12345.

[Checksum Error] Packet 3 invalid. Sending NAK.

[Flow Control] Updated receive window: [4, 5, 6, 7, 8].
```