Resumo Redes

Resumo tópicos da Prova 2

Junho 2025
26.06.2025

INFORMAÇÕES

Baseado nos materiais de aula

AUTOR

Enzo Serenato
enzoserenato@gmail.com
Ponta Grossa, Paraná

Sumário

1. Comutação

1.1. Comutação de circuitos

A comutação de circuitos é um método de comunicação em redes no qual um caminho dedicado é estabelecido entre os dispositivos antes da transmissão de dados. Esse caminho permanece reservado exclusivamente para a comunicação até que a transmissão seja concluída. Em redes telefônicas analógicas tradicionais, um circuito físico é mantido ativo durante toda a conversa, independentemente de haver transmissão contínua de dados. O canal não é compartilhado com outros usuários.

Vantagens:

- Garantia de largura de banda.
- Baixa latência.

Desvantagem:

- Ineficiência no uso de recursos, pois o circuito permanece reservado mesmo quando não há transmissão de dados.
- Requer softwares complexos de sinalização para coordenar comutadores ao longo do caminho.

Multiplexação:

- **FDM (Multiplexação por Divisão de Frequência)**: Divide o espectro de frequência do enlace em bandas, alocando uma banda para cada conexão (ex.: 4 kHz em redes telefônicas).
- **TDM (Multiplexação por Divisão de Tempo)**: Divide o tempo em quadros fixos com compartimentos (slots), reservando um slot por quadro para cada conexão. A taxa de transmissão de um circuito é a taxa do quadro vezes o número de bits por slot (ex.: 8 mil quadros/s com slots de 8 bits = 64 kbits/s).

Exemplo numérico:

- Para enviar um arquivo de 640 kbits por uma rede TDM com 24 slots e taxa de enlace de 1,536 Mbits/s, cada circuito tem taxa de 64 kbits/s. O tempo de transmissão é 10 segundos, mais 500 ms para ativação do circuito, totalizando 10,5 segundos, independente do número de enlaces.

1.2. Comutação de pacotes

A comutação de pacotes, inventada nos anos 1960, consiste em dividir mensagens em pequenos blocos chamados pacotes, que contêm a identificação do destinatário. Vários transmissores compartilham os enlaces da rede, e os roteadores encaminham os pacotes aos destinatários. Computadores ociosos não ocupam recursos da rede, promovendo um compartilhamento eficiente.

Vantagens:

- Compartilhamento eficiente dos recursos da rede.
- Flexibilidade no uso da infraestrutura.
- Permite suportar mais usuários que a comutação de circuitos, especialmente em tráfego esporádico (ex.: 35 usuários com 10% de atividade podem compartilhar um enlace de 1 Mbit/s com probabilidade de congestionamento de apenas 0,0004).

Desvantagens:

- Congestionamentos podem aumentar a latência.

- Possibilidade de perda de pacotes.

Transmissão armazena-e-reenvia:

- Comutadores de pacotes recebem o pacote inteiro antes de transmiti-lo ao próximo enlace, causando atraso de transmissão (ex.: pacote de L bits em enlace de R bits/s leva L/R segundos).
- Para N enlaces, o atraso fim a fim de um pacote é N **(L/R). Para P pacotes, o atraso total é (N + P - 1)** (L/R).

Atrasos de fila e perda:

- Pacotes esperam em buffers de saída se o enlace estiver ocupado, causando atrasos de fila variáveis.
- Buffers finitos podem levar à perda de pacotes se lotados.

	Circuitos	**Pacotes**
Linha	Caminho dedicado, reservado exclusivamente	Enlaces compartilhados por pacotes
Recursos	Ineficiente, reservado mesmo sem uso	Eficiente, usado sob demanda
Latência	Baixa, garantida	Variável, sujeita a congestionamentos
Perda	Nenhuma, conexão estável	Possível devido a congestionamentos

2. Protocolo de comunicação de dados

Um protocolo de comunicação define o formato, a ordem das mensagens trocadas e as ações a serem tomadas pelas entidades, incluindo o tratamento de erros. Ele especifica desde detalhes de baixo nível, como voltagem na transmissão, até questões de alto nível, como o formato e a semântica das mensagens de uma aplicação. Protocolos são organizados em uma pilha de camadas, onde cada camada tem uma função distinta, e as camadas correspondentes entre dispositivos se comunicam. A interoperabilidade é garantida por padrões, evitando dependência de fabricantes ou provedores.

Pilha de protocolos, cabeçalhos e encapsulamento: Os dados passam por várias camadas, onde cada uma adiciona um cabeçalho com informações específicas (encapsulamento). Por exemplo:

- **Camada de Aplicação**: Gera a mensagem (ex.: HTML).
- **Camada de Transporte**: Adiciona cabeçalho TCP (números de porta, sequência).
- **Camada de Rede**: Adiciona cabeçalho IP (endereços IP, TTL).
- **Camada de Enlace**: Adiciona cabeçalho Ethernet (endereços físicos, CRC).

O processo inverso ocorre no destino, onde cada camada remove seu cabeçalho.

Vantagens da arquitetura em camadas:

- Modularidade: Facilita modificações em uma camada sem afetar as demais, desde que o serviço oferecido permaneça o mesmo.

- Simplificação: Permite discutir e projetar partes específicas de um sistema complexo.

Desvantagens:
- Possível duplicação de funcionalidades entre camadas (ex.: recuperação de erros no enlace e fim a fim).
- Dependência de informações de outras camadas, violando a separação (ex.: necessidade de carimbos de tempo).

3. Modelo OSI

O Modelo OSI (Open Systems Interconnection) é um padrão que organiza a comunicação em redes em sete camadas, cada uma com funções específicas:

1. **Física**: Define o meio de transmissão e propriedades elétricas (ex.: frequências, sinais).
2. **Enlace de Dados**: Gerencia a comunicação com a rede física, incluindo endereçamento físico e acesso ao meio.
3. **Rede**: Define a comunicação entre redes interconectadas, como endereçamento IP.variable and unpredictable end-to-end delays due to variable and unpredictable queuing delays).
4. **Transporte**: Garante comunicação confiável entre processos, com controle de fluxo e congestionamento.
5. **Sessão**: Gerencia sessões entre aplicações, incluindo delimitação e sincronização da troca de dados.
6. **Apresentação**: Trata da formatação, compressão, codificação e criptografia dos dados.
7. **Aplicação**: Define como aplicações se comunicam (ex.: HTTP, SMTP).

Comparação com a pilha da Internet:
- A pilha da Internet tem cinco camadas: Física, Enlace, Rede, Transporte e Aplicação.
- As funções das camadas de Sessão e Apresentação do OSI são incorporadas na camada de Aplicação da Internet, deixando ao desenvolvedor a responsabilidade de implementar esses serviços, se necessário.

Vantagens dos padrões:
- Interoperabilidade entre sistemas de diferentes fabricantes.
- Facilita o desenvolvimento e manutenção de redes.

Desvantagens:
- Complexidade adicional devido à padronização.
- Possível rigidez para inovações fora do padrão.

3.1. Unidade de dados de protocolo (PDU)

As PDUs são as unidades de dados trocadas em cada camada da pilha de protocolos:
- **Camada de Aplicação**: Mensagem (ex.: HTML).
- **Camada de Transporte**: Segmento (ex.: TCP com cabeçalho).
- **Camada de Rede**: Pacote ou Datagrama (ex.: IP com cabeçalho).
- **Camada de Enlace**: Quadro (ex.: Ethernet com cabeçalho).

4. Protocolo HTTP e web

O *Hypertext Transfer Protocol* (HTTP) foi criado para permitir que clientes e servidores troquem informações de maneira prática e eficiente.

4.1. Clientes e servidores web

O HTTP é um protocolo baseado em trocas de mensagens de requisições e respostas entre clientes e servidores. No modo direto de acesso (sem proxy, por exemplo) o agente do cliente abre uma conexão TCP com o servidor web e através dessa conexão envia mensagens de texto (ASCII) com requisições. O cliente também envia ao servidor informações de negociação da transferência, como o idioma preferido por ele, a codificação esperada, se aceita compactação dos dados, os tipos de arquivo aceitos e se a conexão deve ser mantida ou encerrada após a transferência. Essas informações são acrescentadas na requisição, também em forma de texto, nas linhas de cabeçalho.

É importante lembrar que todas essas informações são enviadas por conexões TCP cujo modelo de comunicação é o de sequências de bytes, uma em cada direção.

De forma bem simplificada, o navegador web executa os seguintes passos para acessar uma página web estática:

1. Interpreta a URL fornecida pelo usuário
2. Identifica o nome do servidor contido na URL
3. Obtém o endereço IP do servidor através de uma consulta ao DNS
4. Abre uma conexão TCP com o servidor daquele endereço IP na porta 80 (padrão)
5. Usa o protocolo HTTP para enviar a solicitação de um arquivo HTML ao servidor
6. Recebe o arquivo HTML.
7. Interpreta o arquivo HTML e solicita aos respectivos servidores todos os objetos descritos em URLs contidas no arquivo HTML
8. Apresenta a página para o usuário

4.1.1. Conexões persistentes

Cliente e servidor podem manter as conexões TCP abertas para a solicitação e envio de vários objetos. Essas conexões são chamadas de conexões persistentes. Manter a conexão evita o atraso de abertura de novas conexões e pode superar a lentidão do controle de congestionamento do TCP (o TCP inicia a conexão transmitindo com uma taxa pequena e aumenta-a a medida que mais dados são enviados).

A técnica de *pipelining* permite ao cliente enviar múltiplas requisições numa mesma conexão TCP enquanto aguarda as respostas.

Menos conexões TCP significam uma menor latência e também menos uso de memória nos servidores e clientes e menos uso de CPU pelas partes. Porém, o problema de usar uma única conexão TCP para obter muitos objetos é que o atraso de quem está na frente da fila atrasará todos os envios subsequentes devido ao caráter sequencial da comunicação usando TCP.

A versão 2.0 do protocolo HTTP tenta resolver esse problema criando uma camada intermediária de comunicação que ao invés de utilizar o canal TCP diretamente para enviar os objetos, faz a multiplexação do canal quebrando as mensagens em pedaços pequenos e as enviando de forma intercalada. Assim, espera-se que aplicações web que utilizem a versão 2.0 do protocolo necessitem abrir um número menor de conexões TCP.

4.2. Formato das mensagens

As linhas das mensagens HTTP são separadas por uma sequência de dois caracteres especiais: *carriage return* (CR) e *line feed* (LF).

A resposta do servidor contém na primeira linha o status da resposta composto de um número e uma sentença legível que identificam o status. Exemplos de linhas de status são: “200 OK”, “404 Not Found”, “301 Moved Permanently” e “304 Not Modified”. Após a linha de status, seguem linhas contendo os cabeçalhos da resposta, seguidas de uma linha em branco e os bytes que compõe a entidade enviada, caso exista.

4.3. Cabeçalhos HTTP

Alguns tipos de cabeçalhos:

- Content-Length: tamanho do documento em octetos (bytes)
- Content-Type: tipo do documento
- Content-Encoding: codificação do documento
- Content-Language: idioma do documento

Como o HTTP suporta aquivos binários, não é possível a utilização de caracteres ou sequências especiais para delimitar o final dos objetos. Há duas maneiras de resolver o problema.

A primeira é usar o cabeçalho Content-Length para especificar o número de bytes do objeto. A segunda é usar um conteúdo chamado chunked.

No método de transferência chunked, o objeto sendo transferido pode ser dividido em partes (chunks). Cada parte começa com uma linha contendo unicamente um número em ASCII que representa o número de bytes da parte. Seguem então os bytes daquela parte. Um número zero como tamanho da parte indica o fim do objeto.

Outros exemplos de cabeçalhos:

- Connection: close

4.3.1. Exemplo: Negociação: do servidor, do navegador

```
Accept: text/html, text/plain; q=0.5, text/x-dvi; q=0.8
```

Requisições Condicionais:

```
If-Modified-Since: Mon, 01 Apr 2013 05:00:01 GMT
```

Esse cabeçalho é utilizado para o cliente informar o servidor qual é a data do objeto armazenado em seu cache. Caso o objeto não tenha sido modificado, ele não é devolvido pelo servidor e o cliente deve usar o objeto que está no cache.

4.4. Cookies

O protocolo HTTP não mantém o estado de conexões (*stateless*) e isso significa que cada requisição do cliente deve conter informações completas sobre o que ele está requisitando. Sites que precisam guardar informações sobre os usuários que os navegam, como sites onde os usuários fazer login, tão comuns hoje em dia, requerem a utilização de um recurso previsto no HTTP e chamado de cookies.

Cookies funcionam da seguinte maneira: quando o cliente acessa o site pela primeira vez, o servidor envia para ele um cabeçalho chamado set-cookie seguido de uma string que é o cookie. Quando o cliente faz uma nova requisição ao mesmo site, o cookie é

enviado para o servidor em um cabeçalho de nome cookie. Dessa maneira, o cookie pode conter a identificação do usuário.

Cabe a aplicação web armazenar as informações de credencias dos usuários e todas as informações relativas ao contexto do acesso do usuário ao site, não sendo esse armazenamento responsabilidade do HTTP.

É importante que sites que utilizem cookies para identificar usuários usem conexões seguras para que a criptografia garanta que os cookies não sejam interceptados e usados por terceiros para se passarem pelo usuário do site. Essa utilização malicioso dos cookies é chamada de sequestro de sessão.

4.5. Proxies web

O HTTP descreve o suporte para cache de dados e para proxies web. Os proxies web são programas intermediários entre clientes e servidores cuja principal função é fazer cache de dados e reduzir o tráfego de dados no enlace de acesso do cliente.

Além do método GET, o cliente pode enviar solicitações utilizando um dos métodos a seguir: HEAD, POST, PUT, POST, DELETE e OPTIONS. O método HEAD é análogo ao GET, porém o servidor devolve apenas o cabeçalho, sem devolver o objeto associado à requisição. O método POST permite o envio de dados para serem processados pelo servidor. O método PUT serve para fazer UPLOAD de uma representação para uma URI. O comando DELETE serve para remover um recurso do servidor. O método OPTIONS devolve uma lista com os métodos aceitos pelo servidor.

4.6. Versões

Versões 1, 1.1, 2, 3.

O HTTP foi inicialmente concebido por Tim Berners-Lee em 1989 no CERN, junto com o HTML, com o objetivo de transferir arquivos, buscar índices de documentos e permitir negociações entre clientes e servidores. Antes do HTTP, protocolos como FTP (1971) e SMTP (1981) eram usados para transferência de arquivos e e-mails na ARPANET e na Internet. A primeira implementação, HTTP 0.9 (1990-1991), era simples, suportando apenas requisições GET para documentos HTML, com respostas em ASCII e conexões fechadas após cada transferência.

Com o crescimento da Web, o protocolo evoluiu:

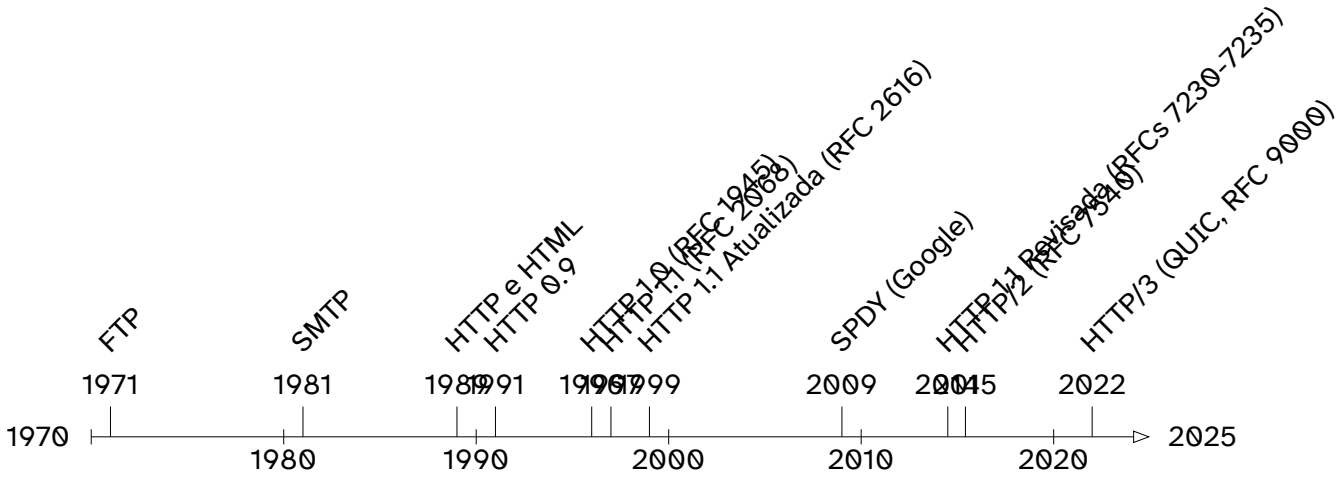
- HTTP 1.0 (1996): Formalizado na RFC 1945, introduziu métodos como POST e HEAD, cabeçalhos para metadados, negociação de tipo de mídia e suporte a proxies, mas usava conexões não persistentes por padrão.
- HTTP 1.1 (1997-1999): Definido na RFC 2068 e atualizado na RFC 2616, trouxe conexões persistentes, pipelining, autenticação, transferência chunked e caching avançado.

Em 2014, foi revisado e dividido em seis RFCs (7230 a 7235), refinando aspectos como sintaxe, semântica e caching.

- HTTP/2 (2015): Baseado no SPDY (2009) do Google e formalizado na RFC 7540, focou em eficiência com multiplexação, compressão de cabeçalhos, push do servidor e enquadramento binário, reduzindo a latência e o uso de conexões TCP.
- HTTP/3 (2022): Utiliza o QUIC sobre UDP (RFC 9000, 2021), oferecendo multiplexação sem bloqueio de linha, criptografia obrigatória, conexões rápidas (0-RTT/1-

-RTT) e suporte a mobilidade, resolvendo limitações do TCP como o bloqueio head-of-line.

A Web, inicialmente limitada a texto e imagens, passou a suportar multimídia, scripts e páginas dinâmicas, transformando a Internet em uma rede essencial para comunicação e comércio.



4.6.1. Tecnologias do HTTP e Sua Evolução

As tecnologias incorporadas ao HTTP ao longo de suas versões refletem a necessidade de maior eficiência, segurança e suporte a aplicações web modernas.

- **Métodos HTTP**: Métodos como GET, POST e HEAD (HTTP 1.0) permitiam operações básicas de recuperação e envio de dados. O HTTP 1.1 expandiu com PUT, DELETE, OPTIONS, TRACE e CONNECT, habilitando aplicações RESTful e túneis seguros (HTTPS).
- **Conexões Persistentes**: Introduzidas no HTTP 1.1 com o cabeçalho `Connection: keep-alive`, reduziram a sobrecarga de abrir/fechar conexões TCP. O HTTP/2 aprimorou isso com multiplexação, e o HTTP/3, via QUIC, adicionou reconexão rápida (0-RTT/1-RTT) e suporte a redes móveis, minimizando latência em cenários dinâmicos.
- **Pipelining**: Disponível no HTTP 1.1, permitia enviar múltiplas requisições sem esperar respostas, mas sofria com bloqueio head-of-line (HOL). O HTTP/2 substituiu o pipelining por multiplexação, e o HTTP/3 eliminou o HOL completamente com QUIC, garantindo processamento paralelo eficiente.
- **Multiplexação**: Introduzida no HTTP/2, permite múltiplos fluxos bidirecionais em uma única conexão TCP, usando enquadramento binário. O HTTP/3 aprimorou com QUIC sobre UDP, eliminando HOL no nível de transporte e aumentando a resiliência em redes instáveis.
- **Compressão de Cabeçalhos**: O HTTP/2 implementou o HPACK, que usa tabelas dinâmicas e codificação Huffman para reduzir o tamanho dos cabeçalhos, diminuindo latência. O HTTP/3 usa QPACK, adaptado para QUIC, mantendo eficiência e adicionando resiliência a perdas de pacotes.
- **Push do Servidor**: Introduzido no HTTP/2, permite que servidores enviem recursos (ex.: CSS, JS) proativamente, reduzindo latência. O HTTP/3 mantém essa

funcionalidade, otimizada com QUIC para entrega rápida, especialmente em páginas web complexas.

- **Protocolo de Transporte**: O HTTP 1.0 e 1.1 usam TCP, que, embora confiável, sofre com latência em conexões iniciais e HOL. O HTTP/2 otimiza o TCP com enquadramento binário, enquanto o HTTP/3 adota QUIC sobre UDP, integrando criptografia, multiplexação e conexões rápidas, ideal para redes modernas.
- **Caching**: O HTTP 1.0 oferecia caching básico com `Last-Modified`. O HTTP 1.1 introduziu `ETag`, `Cache-Control` e `Expires`, otimizando reuso de recursos. O HTTP/2 e HTTP/3 mantêm esses mecanismos, com push do servidor e QUIC reduzindo a necessidade de requisições repetitivas.
- **Criptografia**: Inicialmente opcional (SSL no HTTP 1.0, TLS no 1.1), tornou-se recomendada no HTTP/2 com TLS 1.2 e obrigatória no HTTP/3 com QUIC e TLS 1.3.
- **Transferência Chunked**: Introduzida no HTTP 1.1, permite streaming de dados em blocos, ideal para respostas dinâmicas. O HTTP/2 e HTTP/3 mantêm essa funcionalidade, com enquadramento binário e QUIC.

Feature	HTTP 1.0	HTTP 1.1	HTTP/2	HTTP/3
Métodos	GET, POST e HEAD	PUT, DELETE, OPTIONS, TRACE e CONNECT. Suporta APIs RESTful.	Mesmos métodos do 1.1	Mesmos métodos do 2.0
Conexões Persistentes	Não suportado. Cada requisição abre e fecha uma conexão TCP	Introduz `Connection: keep-alive`, permitindo múltiplas requisições na mesma conexão TCP	Melhora persistência com multiplexação	Persiste conexões via QUIC, com reconexão rápida (0-RTT/1-RTT)
Pipelining	Não suportado	Permite enviar múltiplas requisições sem esperar respostas, mas sofre com bloqueio head-of-line (HOL), atrasando obje-	Substituído por multiplexação, que elimina HOL ao processar requisições/respostas em paralelo via fluxos independentes.	Não usa pipelining. Multiplexação via QUIC elimina HOL completamente

Feature	HTTP 1.0	HTTP 1.1	HTTP/2	HTTP/3
		tos subsequentes.		
Multiplexação	Não suportado. Cada requisição é processada sequencialmente	Não suportado. Pipelining tenta mitigar, mas HOL persiste	Sim, via fluxos bidirecionais em uma única conexão TCP, permitindo processamento paralelo de req/res. Usa enquadramento binário.	Multiplexação aprimorada com QUIC sobre UDP, eliminando HOL no nível de transporte e suportando fluxos independentes
Compressão de Cabeçalhos	Não suportado. Cabeçalhos em texto puro (ASCII)	Não suportado. Cabeçalhos continuam em texto puro	Usa HPACK, com tabelas dinâmicas e codificação Huffman	Usa QPACK, mantendo compressão com tabelas dinâmicas e codificação Huffman
Push do Servidor	Não suportado. Clientes devem solicitar todos os recursos explicitamente	Não suportado. Mesma limitação do 1.0	Permite que o servidor envie recursos proativamente antes da solicitação do cliente	Mantém push do servidor, mas otimizado com QUIC
Protocolo de Transporte	Usa TCP na porta 80, com conexão estabelecida para cada req	Usa TCP, com conexões persistentes na porta 80 ou 443 (HTTPS)	Usa TCP (com TLS na 443), com enquadramento binário e multiplexação	Usa QUIC sobre UDP (443), com criptografia integrada, multiplexação sem HOL e suporte a conexões rápidas (0-RTT).

Feature	HTTP 1.0	HTTP 1.1	HTTP/2	HTTP/3
Caching	Suporte básico com `Last-Modified` e `If-Modified-Since`	Avançado com `ETag`, `Cache-Control`, `Expires` e validação condicional	Mesmos mecanismos do 1.1, mas otimizados com push do servidor e multiplexação	Mesmos mecanismos, mas com QUIC
Criptografia	Opcional, via SSL inicial (HTTPS na porta 443), mas não amplamente utilizado	Opcional, com TLS substituindo SSL. HTTPS ganha adoção, mas muitos sites ainda usam HTTP puro.	Recomenda TLS 1.2 ou superior, com HTTPS como padrão na maioria dos navegadores	Criptografia obrigatória via QUIC, integrada ao protocolo, com TLS 1.3
Transferência Chunked	Não suportado. Conteúdo é enviado com tamanho fixo via `Content-Length`	Introduz transferência chunked, permitindo envio de dados em blocos de tamanho variável	Mantém chunked, otimizado com enquadramento binário	Mantém chunked com QUIC

5. SSH

O **Secure Shell (SSH)** é um protocolo de comunicação segura que proporciona login remoto seguro, execução de comandos em hosts remotos e transferência de arquivos, substituindo protocolos inseguros como Telnet, rlogin e rsh. Ele garante sigilo, integridade de dados, autenticação e detecção de ataques, conforme definido na RFC 4251 e complementado por outras RFCs.

5.1. Características do SSH

- **Sigilo**: Garante que os dados transmitidos sejam confidenciais, utilizando criptografia.
- **Integridade de Dados**: Detecta alterações não autorizadas nas mensagens.
- **Autenticação**: Verifica a identidade das partes envolvidas, geralmente por chaves públicas ou senhas.
- **Detecção de Ataques**: Identifica tentativas de comprometimento, como interceptação ou falsificação.

5.2. Usos do SSH

O SSH é amplamente utilizado para:

- **Login Remoto Seguro**: Substitui Telnet e rlogin, permitindo acesso seguro a máquinas remotas.
- **Execução de Comandos Remotos**: Substitui rsh, possibilitando a execução de comandos em hosts remotos (ex.: `ssh user@host comando`).
- **Transferência de Arquivos Segura**: Usado com ferramentas como SCP e SFTP.
- **Aplicações de Backup e Espelhamento**: Integração com ferramentas como rsync e unison.
- **Tunelamento e Encaminhamento**: Cria túneis seguros para tráfego de rede (ex.: `ssh -L porta_local:host_dest:porta user@host`).
- **Navegação via Proxy Criptografado**: Utiliza SOCKS para navegação segura (ex.: `ssh -D porta user@servidor`).
- **Implementação de VPN**: Encaminha tráfego de rede de forma segura.
- **Redirecionamento Gráfico**: Suporta aplicações gráficas no Linux com `ssh -X`.
- **Montagem de Diretórios Remotos**: Usa sshfs para acessar diretórios remotos como se fossem locais.

5.3. Autenticação com Chaves Públicas

O SSH suporta autenticação por chaves públicas, mais segura contra ataques de força bruta do que login com senha:

- **Geração de Chaves**: Usa o comando `ssh-keygen` para criar um par de chaves (pública e privada). Exemplo:

  ```
  ssh-keygen -t rsa -b 4096 -f ~/.ssh/nome-da-chave
  ```

  - Tipos de chaves suportadas: RSA, DSA, ECDSA, Ed25519.
  - Os arquivos gerados ficam no diretório `~/.ssh`.
- **Chaves Autorizadas**: A chave pública é copiada para o arquivo `~/.ssh/authorized_keys` na máquina remota com:

  ```
  ssh-copy-id -i ~/.ssh/nome-da-chave user@host
  ```
- **Login Automático**: Permite autenticação sem senha, utilizando a chave privada.

5.4. Configuração e Automatização

- **ssh-agent**: Gerencia chaves privadas para autenticação automática.
- **Arquivo de Configuração**: O arquivo `~/.ssh/config` simplifica conexões, definindo parâmetros como host, usuário, porta e chave. Exemplo:

  ```
  Host home
      HostName example.com
      User apollo
      Port 4567
      IdentityFile ~/.ssh/nome-da-chave
      ServerAliveInterval 100
      ServerAliveCountMax 3
  ```
- **Tunelamento no Config**: Configura túneis automáticos, como:

  ```
  Host server-name
      ProxyCommand ssh hostname nc via-this-host 22
      User jaime
  ```

5.5. Tunelamento

O SSH permite criar túneis para redirecionar tráfego de forma segura. Exemplo de tunelamento local:

```
ssh -f -N -L 3310:localhost:3306 servidor
```

Nesse caso, a porta local 3310 é mapeada para a porta 3306 do servidor remoto, permitindo, por exemplo, acesso seguro a um banco de dados MySQL:

```
mysqldump -P 3310 -h 127.0.0.1 <opções>
```

5.6. Navegação por Proxy SOCKS

O SSH pode configurar um proxy SOCKS para navegação segura:

```
ssh -C -N -f -o ServerAliveInterval=100 -o ServerAliveCountMax=3 -D 5003
user@servidor
```

- `-D`: Define a porta do proxy SOCKS.
- `-C`: Habilita compressão.
- `-N -f`: Executa em segundo plano sem comandos remotos.

5.7. Integração com SSL/TLS

Embora o SSH seja independente, ele complementa protocolos como SSL/TLS, que também garantem autenticação, sigilo e integridade. O SSL/TLS, descrito no documento, utiliza:

- **Autenticação por Chave Pública**: Via certificados digitais.
- **Criptografia Simétrica**: Ex.: AES-CBC para transmissão de dados.
- **Código de Autenticação de Mensagens (MAC)**: Garante integridade.
- **Forward Secrecy**: Protege comunicações anteriores.
- **Tipos de Mensagens**: Handshake, mudança de algoritmos, aplicação, alerta e heartbeat.
- **Alertas**: Incluem erros como MAC incorreto ou problemas no certificado.

Funcionalidade	Descrição
Login Remoto	Acesso seguro a máquinas remotas, substituindo Telnet/rlogin
Execução de Comandos	Permite executar comandos em hosts remotos (ex.: `ssh user@host comando`)
Transferência de Arquivos	Suporte a SCP, SFTP e ferramentas como rsync
Tunelamento	Redireciona tráfego por túneis seguros (ex.: `ssh -L`)
Proxy SOCKS	Navegação segura via proxy (ex.: `ssh -D`)
Autenticação	Suporta chaves públicas (RSA, ECDSA, etc.) e senhas, com configuração via `~/.ssh/authorized_keys`

Funcionalidade	Descrição
Automatização	Uso de `ssh-agent` e arquivo `~/.ssh/config` para conexões automáticas

6. FTP

O **File Transfer Protocol (FTP)** é um protocolo de aplicação projetado para transferência de arquivos entre um cliente e um servidor em uma rede, permitindo download e upload de arquivos em qualquer formato. Ele suporta autenticação, permissões, navegação de diretórios e é independente de sistema operacional e hardware. Definido em RFCs públicas, o FTP opera sobre o protocolo TCP, utilizando duas conexões distintas: uma para controle e outra para dados.

6.1. Características do FTP

- **Transferência Bidirecional**: Permite download (get) e upload (put) de arquivos.
- **Formatos Flexíveis**: Suporta qualquer tipo de arquivo (binário ou texto).
- **Autenticação e Permissões**: Requer login (usuário e senha) e suporta controle de acesso.
- **Navegação de Diretórios**: Permite listar e navegar pelos diretórios do servidor.
- **Independência**: Funciona em diferentes sistemas operacionais e arquiteturas de hardware.

6.2. Conexões no FTP

O FTP utiliza duas conexões TCP separadas:

- **Conexão de Controle**:
  - Iniciada pelo cliente na porta padrão 21 do servidor.
  - Usada para enviar comandos (ex.: listar diretórios, solicitar arquivos) e receber respostas do servidor.
  - Permanece ativa durante toda a sessão.
- **Conexão de Dados**:
  - Geralmente iniciada pelo servidor a partir da porta 20 para uma porta dinâmica no cliente.
  - Usada para transferir arquivos ou listas de diretórios.
  - Aberta e fechada para cada transferência.

Modo Passivo:

- Usado para contornar problemas com NAT (Network Address Translation), onde o cliente inicia tanto a conexão de controle quanto a de dados, e o servidor aloca uma porta dinâmica para os dados.

6.3. Interação Típica

A interação entre cliente e servidor segue estas etapas:

1. O cliente abre uma conexão de controle com o servidor (porta 21).
2. O cliente envia comandos, como solicitação de listagem de diretórios.
3. O servidor abre uma conexão de dados (porta 20 para uma porta dinâmica no cliente).
4. O servidor envia a lista de arquivos ou o arquivo solicitado.
5. A conexão de dados é fechada após a transferência.

6. Para download, o cliente envia um comando `get`, e o servidor abre uma nova conexão de dados para enviar o arquivo.
7. Após a conclusão, o cliente envia o comando `QUIT` na conexão de controle e fecha a sessão.

6.4. Portas

- **Porta de Controle**: 21 (padrão no servidor).
- **Porta de Dados**: 20 (padrão no servidor, para conexões ativas); em modo passivo, usa portas dinâmicas.
- **Cliente**: Utiliza portas dinâmicas (alocadas pelo sistema operacional).

6.5. Considerações

- **Segurança**: O FTP padrão não utiliza criptografia, tornando-o vulnerável a interceptações. Para segurança, recomenda-se o uso de FTPS (FTP com SSL/TLS) ou SFTP (baseado em SSH).
- **NAT**: O modo passivo é frequentemente usado para compatibilidade com firewalls e NAT.
- **Eficiência**: A separação entre conexões de controle e dados permite flexibilidade, mas aumenta a complexidade em comparação com protocolos modernos.

Funcionalidade	Descrição
Transferência de Arquivos	Suporta download (get) e upload (put) em qualquer formato
Conexões	Separa controle (porta 21) e dados (porta 20 ou dinâmica em modo passivo)
Autenticação	Requer usuário e senha, com suporte a permissões
Navegação	Permite listar e navegar diretórios no servidor
Independência	Compatível com diferentes sistemas operacionais e hardware
Modo Passivo	Contorna NAT, com cliente iniciando ambas as conexões

7. Correio Eletrônico e os Protocolos SMTP, IMAP, POP3 & MIME

O correio eletrônico é uma aplicação de rede que permite o envio, recebimento e gerenciamento de mensagens entre usuários. Ele opera por meio de agentes de usuário (clientes de e-mail ou webmail) e agentes de transferência de mensagens (MTAs, servidores de e-mail), utilizando protocolos como SMTP, IMAP, POP3 e MIME para diferentes funções.

7.1. Funcionamento Básico

O processo de envio e recebimento de e-mails segue estas etapas:

1. O usuário cria uma mensagem no cliente de e-mail.
2. O cliente coloca a mensagem em uma fila (spool).
3. O MTA seleciona a mensagem da fila e estabelece uma conexão TCP com o servidor do destinatário.
4. O MTA envia a mensagem via SMTP, que é armazenada na caixa postal do destinatário.
5. O destinatário acessa a mensagem usando um cliente de e-mail (via POP3 ou IMAP).

7.2. Funcionalidades

- **Listas de E-mail**: Envio para múltiplos destinatários.
- **Cópia (Cc) e Cópia Oculta (Bcc)**: Inclui destinatários adicionais visíveis ou ocultos.
- **Prioridades**: Define urgência da mensagem.
- **Encriptação**: Protege o conteúdo (ex.: via PGP ou S/MIME).
- **Delegação**: Permite terceiros gerenciarem caixas postais.
- **Encaminhamento e Resposta Automática**: Regras para redirecionamento ou respostas automáticas.
- **Filtro de Spam**: Identifica e separa mensagens indesejadas.

7.3. Formato de Endereço

- Formato: `destinatário@servidor`.
- O domínio (`@servidor`) é resolvido via DNS, que retorna registros MX (Mail Exchanger) indicando os servidores de e-mail. Exemplo:

```
$ dig -t MX gmail.com
gmail.com. 3415 IN MX 10 alt1.gmail-smtp-in.l.google.com.
gmail.com. 3415 IN MX 20 alt2.gmail-smtp-in.l.google.com.
```

7.4. Protocolos

7.4.1. SMTP (Simple Mail Transfer Protocol)

O SMTP é usado para enviar mensagens entre servidores de e-mail e de clientes para servidores. Opera nas portas 25, 465 (com SSL) ou 587 (com TLS).

- **Características**:
  - Protocolo baseado em texto (ASCII).
  - Envia mensagens em conexões TCP, garantindo confiabilidade.
  - Suporta comandos como `HELO`, `MAIL FROM`, `RCPT TO`, `DATA` e `QUIT`.
- **Exemplo de Comunicação**:

```
S: 220 smtp.example.com ESMTP Postfix
C: HELO relay.example.org
S: 250 Hello relay.example.org, I am glad to meet you
C: MAIL FROM:<bob@example.org>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Bob Example" <bob@example.org>
C: To: "Alice Example" <alice@example.com>
C: Subject: Test message
```

```
C:
C: Hello Alice.
C: This is a test message.
C: Your friend, Bob
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
```

7.4.2. POP3 (Post Office Protocol, versão 3)

O POP3 permite que clientes recuperem mensagens de um servidor, geralmente removendo-as após o download. Opera na porta 110 (ou 995 com SSL).

- **Características**:
  - Modelo "download-and-delete": Mensagens são transferidas e excluídas do servidor.
  - Suporta comandos como `APOP` (autenticação), `STAT`, `LIST`, `RETR` (recuperar), `DELE` (deletar) e `QUIT`.
  - Ideal para acesso offline, mas limitado para sincronização em múltiplos dispositivos.
- **Exemplo de Comunicação**:

```
S: +OK POP3 server ready
C: APOP mrose c4c9334bac560ecc979e58001b3e22fb
S: +OK mrose's maildrop has 2 messages (320 octets)
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S: <sends message 1>
S: .
C: DELE 1
S: +OK message 1 deleted
C: QUIT
S: +OK dewey POP3 server signing off
```

7.4.3. IMAP (Internet Message Access Protocol)

O IMAP permite que clientes acessem e gerenciem mensagens diretamente no servidor, mantendo-as sincronizadas. Opera na porta 143 (ou 993 com SSL).

- **Características**:
  - Modelo "sincronizado": Mensagens permanecem no servidor, permitindo acesso de múltiplos dispositivos.
  - Suporta pastas, marcações (ex.: lido/não lido) e busca avançada.
  - Mais complexo que o POP3, mas ideal para uso em dispositivos móveis e webmail.
- **Vantagens sobre POP3**:
  - Sincronização em tempo real.
  - Gerenciamento de mensagens no servidor (ex.: mover, excluir, marcar).

7.4.4. MIME (Multipurpose Internet Mail Extensions)

O MIME estende o formato de e-mails para suportar conteúdo além de texto ASCII, como imagens, áudio, vídeo e anexos. Definido nas RFCs 2045 a 2049.

- **Cabeçalhos MIME**:
  - `MIME-Version`: Indica a versão do MIME (ex.: 1.0).
  - `Content-Type`: Define o tipo de conteúdo (ex.: `text/plain`, `image/jpeg`, `multipart/mixed`).
  - `Content-Transfer-Encoding`: Especifica a codificação (ex.: `base64`, `quoted-printable`, `7bit`).
  - `Content-Description`: Descrição legível do conteúdo.
- **Tipos de Conteúdo**:
  - **text**: `plain`, `html`, `xml`.
  - **image**: `jpeg`, `gif`, `tiff`.
  - **audio**: `mpeg`, `basic`.
  - **video**: `mpeg`, `mp4`.
  - **application**: `pdf`, `zip`, `octet-stream`.
  - **multipart**: `mixed`, `alternative`, `parallel`, `digest`.
- **Codificações**:
  - ASCII (7-bit ou 8-bit).
  - Base64: Codifica binário em caracteres [A-Za-z0-9+/].
  - Quoted-printable: Preserva texto legível com caracteres especiais.
  - Binário: Dados brutos sem codificação.

7.5. Formato de Mensagem (RFC 2822)

As mensagens de e-mail consistem em:

- **Cabeçalho**: Inclui campos como:
  - `To`, `Cc`, `Bcc`: Destinatários.
  - `From`, `Sender`: Remetente.
  - `Subject`: Assunto.
  - `Date`, `Message-Id`, `Reply-To`, `In-Reply-To`, `References`: Metadados.
  - `Received`: Rota dos MTAs.
  - `Return-Path`: Endereço de retorno.
- **Corpo**: Conteúdo da mensagem, formatado com MIME para suportar anexos e multimídia.

7.6. Clientes de E-mail

- **Programas Clientes**: Usam SMTP para enviar e POP3/IMAP para receber e gerenciar mensagens (ex.: Thunderbird, Outlook).
- **Webmail**: Usa protocolos proprietários para acesso via navegador (ex.: Gmail, Outlook Web).

Protocolo/Ferramenta	Descrição
SMTP	Envia mensagens entre servidores e de clientes para servidores (portas 25, 465, 587)
POP3	Recupera mensagens, geralmente as removendo do servidor (porta 110/995)

Protocolo/Ferramenta	Descrição
IMAP	Gerencia mensagens no servidor com sincronização (porta 143/993)
MIME	Estende formato de e-mails para suportar multimídia e anexos (ex.: base64, multipart)
Cabeçalhos	Campos como To, Cc, From, Subject definem metadados da mensagem
Clientes	Programas (ex.: Thunderbird) ou webmail (ex.: Gmail) para acesso e envio

8. Programação com Sockets

A programação com sockets permite comunicação em rede entre cliente e servidor, utilizando protocolos da camada de transporte, como TCP (confiável, orientado a conexão) e UDP (não confiável, sem conexão). Em Python, a biblioteca `socket` é usada para criar, configurar e gerenciar sockets, com suporte a modelos cliente/servidor, servidores não bloqueantes com `select` e concorrência com threads.

8.1. Modelo Cliente/Servidor

- **Cliente**: Inicia a conexão, envia e recebe dados, e fecha a conexão.
  - Passos: Criar socket (`socket()`), conectar ao servidor (`connect()`), enviar/receber dados (`sendall()`/`recv()`), fechar socket (`close()`).
- **Servidor**: Escuta conexões, aceita clientes, processa dados e fecha conexões.
  - Passos: Criar socket (`socket()`), associar endereço/porta (`bind()`), escutar conexões (`listen()`), aceitar conexões (`accept()`), enviar/receber dados, fechar conexão.

8.2. Endpoints

- Um socket TCP é identificado por um par **(endereço IP, porta)**.
- Exemplo:
  - Servidor: `(192.168.1.1, 80)`.
  - Cliente: `(192.168.0.100, 14001)`.

8.3. Cuidados na Programação

- **Envio/Recepção**: Usar laços para garantir que todos os bytes sejam enviados (`sendall()`) ou recebidos (`recv()`).
- **Tratamento de Erros**: Usar blocos `try/except` para capturar erros de socket (ex.: falha na conexão).
- **Fechamento de Conexões**: Verificar se `recv()` retorna vazio, indicando que a conexão foi fechada.

8.4. Criação de Sockets

```
import socket
# Socket TCP
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # IPv4, TCP
# Socket UDP
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)   # IPv4, UDP
s.close()
```

8.5. Acesso ao DNS

A biblioteca `socket` permite resolver nomes de domínio:

```
import socket
dominio = "www.google.com"
ip = socket.gethostbyname(dominio)  # Retorna o IP
info = socket.getaddrinfo(dominio, 80)  # Retorna informações de endereço
```

8.6. Servidor Não Bloqueante com `select`

O módulo `select` permite gerenciar múltiplos sockets sem bloqueio:

- **Função**: `select.select(rlist, wlist, elist, timeout)` verifica sockets prontos para leitura (`rlist`), escrita (`wlist`) ou com erros (`elist`).
- **Estrutura**:

```
rlist, wlist, elist = select.select(readSockets, [], [], 2)
if [rlist, wlist, elist] == [[], [], []]:
    print("Nenhum evento")
else:
    for sock in rlist:
        if sock is acceptSocket:
            dataSocket, addr = sock.accept()  # Novo cliente
            readSockets.append(dataSocket)
        else:
            data = sock.recv(1024)  # Dados de cliente
            if not data:
                sock.close()
                readSockets.remove(sock)
```

8.7. Servidor Concorrente com Threads

Cada cliente é atendido por uma thread separada:

```
import threading
def handle_client(dataSocket, addr):
    while True:
        data = dataSocket.recv(1024)
        if not data:
            dataSocket.close()
            break
        print(data.decode())
# Iniciar thread para novo cliente
dataSocket, addr = acceptSocket.accept()
thread = threading.Thread(target=handle_client, args=(dataSocket, addr))
thread.start()
```

8.8. Aplicação: Monitor de Sistemas

Uma aplicação cliente/servidor que monitora carga de CPU e memória disponível:

- **Cliente**: Usa a biblioteca `psutil` para obter hostname, carga de CPU (`cpu_percent`) e memória disponível (`virtual_memory`). Envia mensagens de 67 bytes a cada 3 segundos:

```
import socket, psutil, time
hostname = socket.gethostname()
uso_cpu = psutil.cpu_percent(interval=1.0)
```

```
mem_livre = psutil.virtual_memory().available * 100 /
psutil.virtual_memory().total
mensagem = f"Hostname:{hostname:10} Carga de CPU:{uso_cpu:5.1f}% Memória
Disponível:{mem_livre:5.1f}%"
sock.sendall(mensagem.encode())
```

- **Servidor**: Recebe mensagens de clientes, usando `select` ou threads para concorrência.
  - **Com `select`** (arquivo `servidor4-select.py`): Gerencia múltiplos clientes sem bloqueio.
  - **Com Threads** (arquivo `servidor5-multithread.py`): Cria uma thread por cliente.
- **Mensagens**: Formato fixo de 67 bytes, tratado com laço para garantir recepção completa:

```
chunks = []
count = 0
while count != 67:
    data = sock.recv(67 - count)
    if not data:
        sock.close()
        break
    chunks.append(data.decode())
    count += len(data)
print("".join(chunks))
```

8.9. Tratamento de Erros

Erros são capturados com blocos `try/except`:

```
try:
    sock.connect((host, port))
except socket.gaierror:
    print("Erro: Endereço do servidor inválido")
except socket.error as e:
    print(f"Erro de conexão: {e}")
finally:
    sock.close()
```

8.10. UDP

O UDP é usado para comunicação sem conexão:

- **Cliente UDP**:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(mensagem.encode(), ("localhost", 12345))
resposta, _ = sock.recvfrom(1024)
```

- **Servidor UDP**:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(("0.0.0.0", 12345))
data, addr = sock.recvfrom(1024)
sock.sendto(resposta.encode(), addr)
```

Funcionalidade	Descrição
Criação de Socket	Usa `socket.socket(AF_INET, SOCK_STREAM)` para TCP ou `SOCK_DGRAM` para UDP
Cliente TCP	Conecta (`connect`), envia (`sendall`), recebe (`recv`), fecha (`close`)
Servidor TCP	Vincula (`bind`), escuta (`listen`), aceita (`accept`), envia/recebe dados
Select	Gerencia múltiplos sockets sem bloqueio com `select.select()`
Threads	Cria uma thread por cliente para concorrência
UDP	Comunicação sem conexão com `sendto` e `recvfrom`
Monitor de Sistemas	Clientes enviam carga de CPU/memória (67 bytes) a cada 3s; servidor usa `select` ou threads

9. Camada de Transporte

A camada de transporte, implementada nos hospedeiros, provê comunicação fim-a-fim entre processos, encapsulando dados de aplicações (ex.: HTTP, SMTP, FTP) em segmentos. Os principais protocolos são TCP (confiável, orientado a conexão) e UDP (não confiável, sem conexão).

9.1. Protocolos de Transporte

- **TCP**: Transmissão confiável, com controle de fluxo e congestionamento, orientada a fluxo de bytes.
- **UDP**: Comunicação sem conexão, com demultiplexação e checagem de erros simples (checksum), entrega mensagens individuais.

9.2. UDP: User Datagram Protocol

- **Características**:
  - Não confiável: permite perdas, duplicações ou entrega fora de ordem.
  - Sem conexão: não requer estabelecimento prévio de conexão.
  - Mensagens individuais: entregues de uma vez, com tamanho limitado.
- **Demultiplexação**: Baseada em endereço IP e porta de destino.
- **Cabeçalho**:
  - Inclui portas de origem e destino, tamanho da mensagem, checksum.

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Source Port*																*Destination Port*															
4	32	*Length*																*Checksum*															
8	64	*Data*																															
12	96																																
⋮	⋮																																

- Exemplo de uso em programação:

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  # UDP
sock.sendto("mensagem".encode(), ("localhost", 12345))
```

9.3. TCP: Transmission Control Protocol

- **Características**:
  - Confiável: garante entrega, ordem correta, sem duplicações.
  - Orientado a conexão: usa handshake de 3 vias para iniciar.
  - Fluxo de bytes: divide dados em segmentos para transmissão eficiente.
- **Propriedades**:
  - Controle de fluxo: evita sobrecarga do receptor.
  - Controle de congestionamento: previne excesso de dados na rede.
  - Sincronização entre transmissor e receptor.

9.4. Cabeçalho TCP

- **Campos principais**:
  - **SrcPort/DstPort**: Identificam portas de origem/destino.
  - **SequenceNum**: Número de sequência do primeiro byte no segmento.
  - **Acknowledgement**: Confirma bytes recebidos no sentido oposto.
  - **AdvertisedWindow**: Tamanho da janela do receptor.
  - **Flags**: SYN, FIN (estabelecimento/fechamento), ACK, RESET, URG, PUSH.
  - **Checksum**: Verifica integridade de cabeçalho, dados e pseudo-cabeçalho.
  - **UrgPtr**: Indica início de dados não urgentes (com flag URG).
- Estrutura:

Offset	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Source Port*																*Destination Port*															
4	32	*Sequence Number*																															
8	64	*Acknowledgement Number (meaningful when ACK bit set)*																															
12	96	*Data Offset*				*Reserved*				*CWR*	*ECE*	*URG*	*ACK*	*PSH*	*RST*	*SYN*	*FIN*	*Window*																
16	128	*Checksum*																*Urgent Pointer (meaningful when URG bit set)*[18]															
20	160	*(Options) If present, Data Offset will be greater than 5.* *Padded with zeroes to a multiple of 32 bits, since Data Offset counts words of 4 octets.*																															
⋮	⋮																																
56	448																																
60	480	*Data*																															
64	512																																
⋮	⋮																																

9.5. Estabelecimento e Finalização de Conexão

- **Handshake de 3 vias**:
  1. Cliente envia SYN.
  2. Servidor responde SYN+ACK.
  3. Cliente envia ACK.
- **Finalização**:
  - Usa flag FIN para encerrar conexão.
  - Estado do cliente: ESTAB → FIN_WAIT_1 → FIN_WAIT_2 → CLOSED.
- Exemplo:

```python
# Cliente TCP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("localhost", 12345))  # Inicia handshake
sock.close()  # Inicia finalização
```

9.6. Janela Deslizante

- **Objetivos**:
  - Garantir entrega confiável e em ordem.
  - Controlar fluxo entre transmissor e receptor.
- **Mecanismo**:
  - Transmissor: `LastByteAcked ≤ LastByteSent ≤ LastByteWritten`.
  - Receptor: `LastByteRead < NextByteExpected ≤ LastByteRcvd + 1`.
  - Usa `SequenceNum` e `Acknowledgement` para rastrear bytes.
- **Controle de Fluxo**:
  - Receptor anuncia `AdvertisedWindow` para limitar transmissões.
  - Exemplo: Receptor com buffer pequeno reduz janela para evitar sobrecarga.

9.7. Confirmações e Retransmissões

- **Confirmações**:
  - Cumulativas: confirmam todos os bytes até `NextByteExpected`.
  - Atraso de até 500 ms para próximo segmento, salvo em casos específicos (ex.: gap detectado, confirmação imediata).
- **Retransmissão Rápida**:
  - Após 3 ACKs duplicados, retransmite segmento com menor número de sequência não confirmado.
  - Exemplo:

```python
# Servidor detecta 3 ACKs duplicados
if acks_duplicados >= 3:
    retransmit_segment(min_seq_not_acked)
```

9.8. Timeouts e RTT

- **RTT (Round-Trip Time)**:
  - Média móvel exponencial: `EstimatedRTT = (1 - α) * EstimatedRTT + α * SampleRTT` ($\alpha = 0.125$).
  - Variação: `DevRTT = (1 - β) * DevRTT + β * |SampleRTT - EstimatedRTT|` ($\beta = 0.25$).
- **Timeout**:
  - `Timeout = EstimatedRTT + 4 * DevRTT`.
  - Evita retransmissões desnecessárias (timeout curto) ou atrasos (timeout longo).

9.9. Controle de Congestionamento

- **Janela de Congestionamento (cwnd)**:
  - Limita bytes em trânsito: `LastByteSent - LastByteAcked < cwnd`.
  - Taxa: `rate = cwnd / RTT` bytes/s.
- **Slow Start**:
  - Inicia com `cwnd = 1 MSS`, dobra a cada RTT até limiar ou perda.
- **Congestion Avoidance**:
  - Aumenta `cwnd` additivamente (+1 MSS por RTT).
  - Reduz à metade após perda.

- **Implementações**:
  - **TCP Tahoe**: Após perda (timeout ou 3 ACKs duplicados), define `cwnd = 1 MSS` e reinicia slow start.
  - **TCP Reno**: Após 3 ACKs duplicados, reduz `cwnd` pela metade e entra em recuperação rápida.

9.10. Problemas da Rede Subjacente

- Perdas, entregas fora de ordem, duplicações, atrasos arbitrários, tamanho limitado de pacotes.
- TCP resolve com confirmações, retransmissões, janela deslizante, e controles de fluxo/congestionamento.

9.11. Integração com Programação de Sockets

- **TCP**: Usa `socket.SOCK_STREAM` para conexões confiáveis.
  - Exemplo: Servidor aceita conexões com `accept()` e gerencia com `select` ou threads.
- **UDP**: Usa `socket.SOCK_DGRAM` para mensagens sem conexão.
  - Exemplo: Cliente envia dados com `sendto()` sem handshake.
- **Aplicação prática**: Monitor de sistemas (ver seção anterior) usa TCP/UDP para enviar métricas de CPU e memória.

Protocolo	Características	Uso em Sockets
TCP	Confiável, orientado a conexão, controle de fluxo/congestionamento	`socket.SOCK_STREAM`, usa `connect`, `accept`, `sendall`, `recv`
UDP	Não confiável, sem conexão, mensagens individuais	`socket.SOCK_DGRAM`, usa `sendto`, `recvfrom`
Janela Deslizante	Garante ordem, confiabilidade, controle de fluxo	Implementado internamente pelo TCP
Controle de Congestionamento	Slow start, congestion avoidance, Tahoe/Reno	Gerenciado pelo sistema operacional