

5954025 - Sistemas Distribuídos

Aula 08b - Comunicação via HTTP: Websockets

Prof. Dr. Denis M. L. Martins

DCM | FFCLRP | USP

Objetivos de Aprendizagem

- Compreender o que é WebSocket a motivação de sua criação.
- Descrever o fluxo de handshake e a estrutura dos frames.
- Diferenciar WebSocket de outras alternativas.
- Implementar uma aplicação simples.

Motivação

- A World Wide Web original foi projetada como um sistema de documentos de hipertexto interconectados
- Embora o HTTP/1.1 tenha introduzido conexões persistentes para reutilização, o modelo de comunicação permaneceu fundamentalmente o mesmo.
- Aplicações modernas exigem resposta imediata
- Atualizações em tempo real são essenciais

Exemplos:

- Chat online
- Jogos multiplayer
- Monitoramento de sistemas

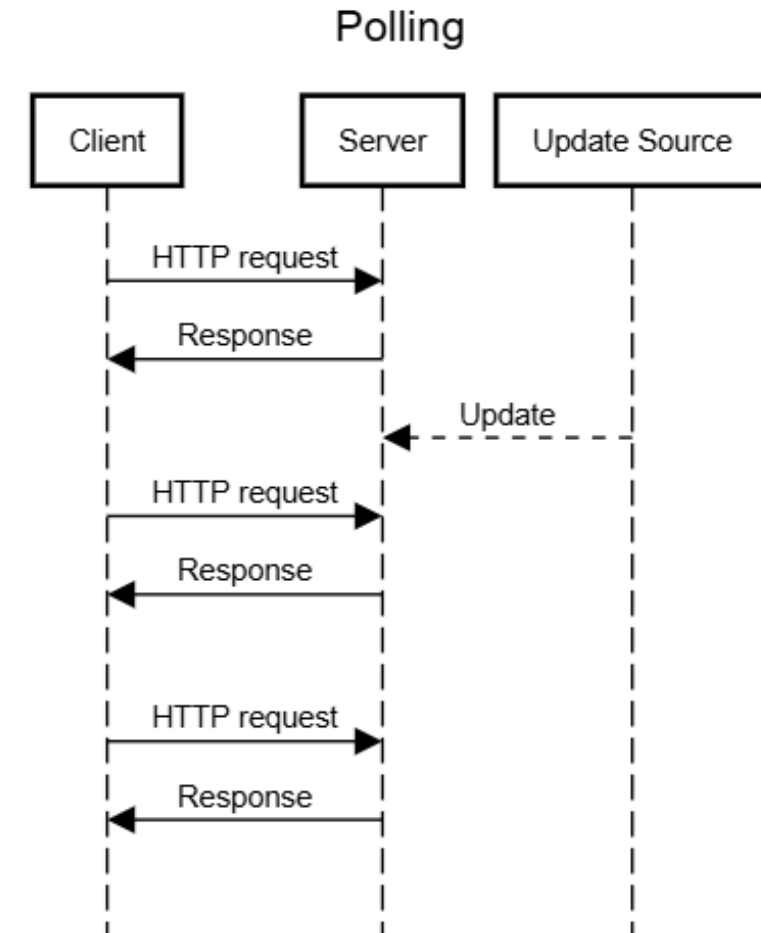
Motivação

- O HTTP é inerentemente *Half-Duplex*, o que significa que o tráfego flui em apenas uma direção por vez.
- Protocolo *stateless*, tratando cada requisição como única e independente, o que exige o envio de cabeçalhos redundantes em todas as interações.
- O overhead é alto: metadados de cabeçalho frequentemente superam o tamanho dos dados reais transmitidos.
- A natureza do HTTP impede que o servidor envie dados de forma proativa para o cliente sem uma requisição pendente
- Aplicações modernas exigem interatividade responsiva e notificações imediatas
 - **Exemplo:** Chat que precisa atualizar constantemente

Soluções Históricas

Polling: Cliente envia requisição a cada intervalo fixo (ex: a cada 10 segundos). Servidor responde com dados acumulados (ou retorna uma mensagem vazia).

- O cliente aguarda o fim da resposta e imediatamente dispara a próxima requisição, mantendo assim uma sequência de chamadas.
- **Latência** controlada pelo intervalo escolhido (quanto menor o intervalo, mais próximo do "tempo real").
- **Sobrecarga de rede e CPU:** requisições frequentes geram tráfego desnecessário, especialmente quando não há dados novos.



Na imagem: diagrama de sequência de Polling por [Jeyeong](#)

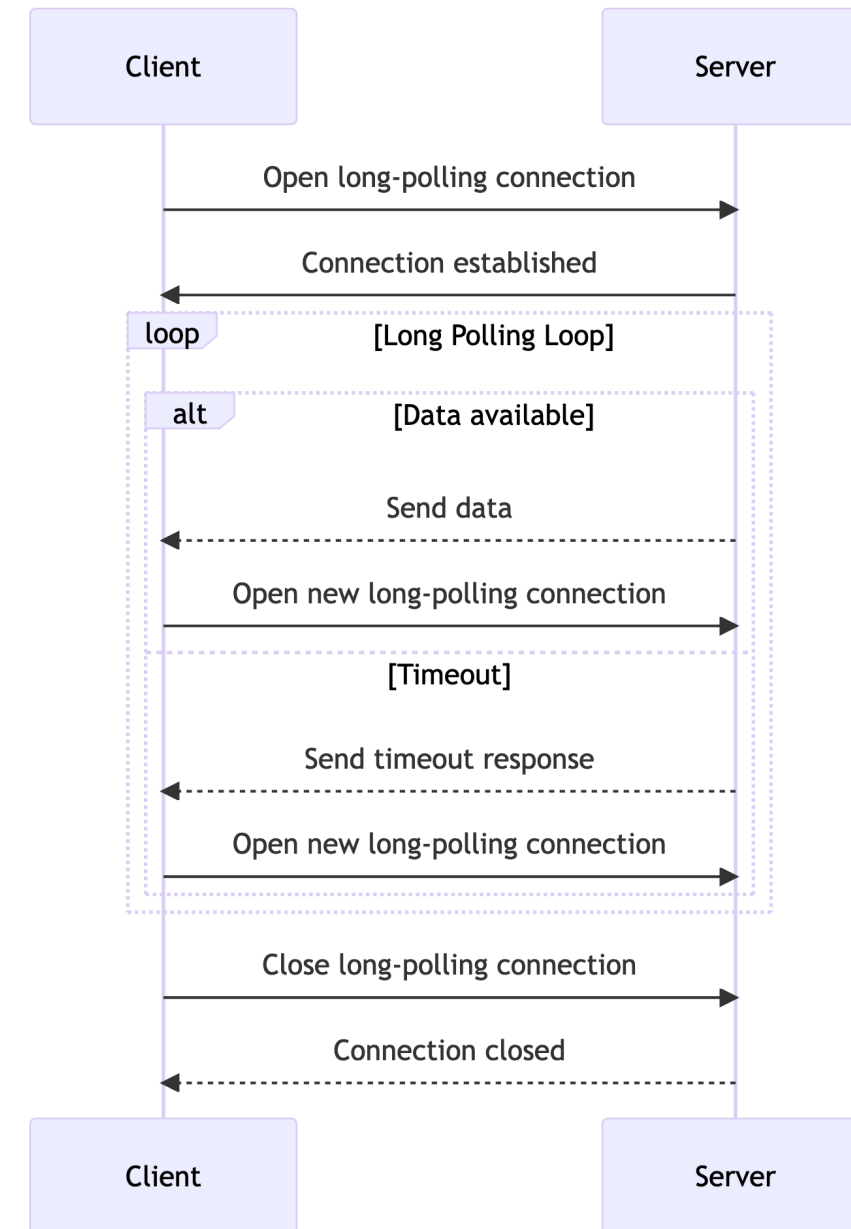
Soluções Históricas (cont.)

Long Polling: Técnica de comunicação web em que o cliente faz uma requisição e mantém a conexão aberta até que o servidor tenha novos dados.

1. Cliente envia request ao servidor.
2. Servidor aguarda (mantém a conexão "aberta").
3. Quando há nova informação, servidor responde e fecha a conexão.
4. Cliente imediatamente abre uma nova requisição para esperar por mais atualizações.

Na imagem: diagrama de sequência de Long Polling. Fonte: [Code](#)

[Magazine](#) 

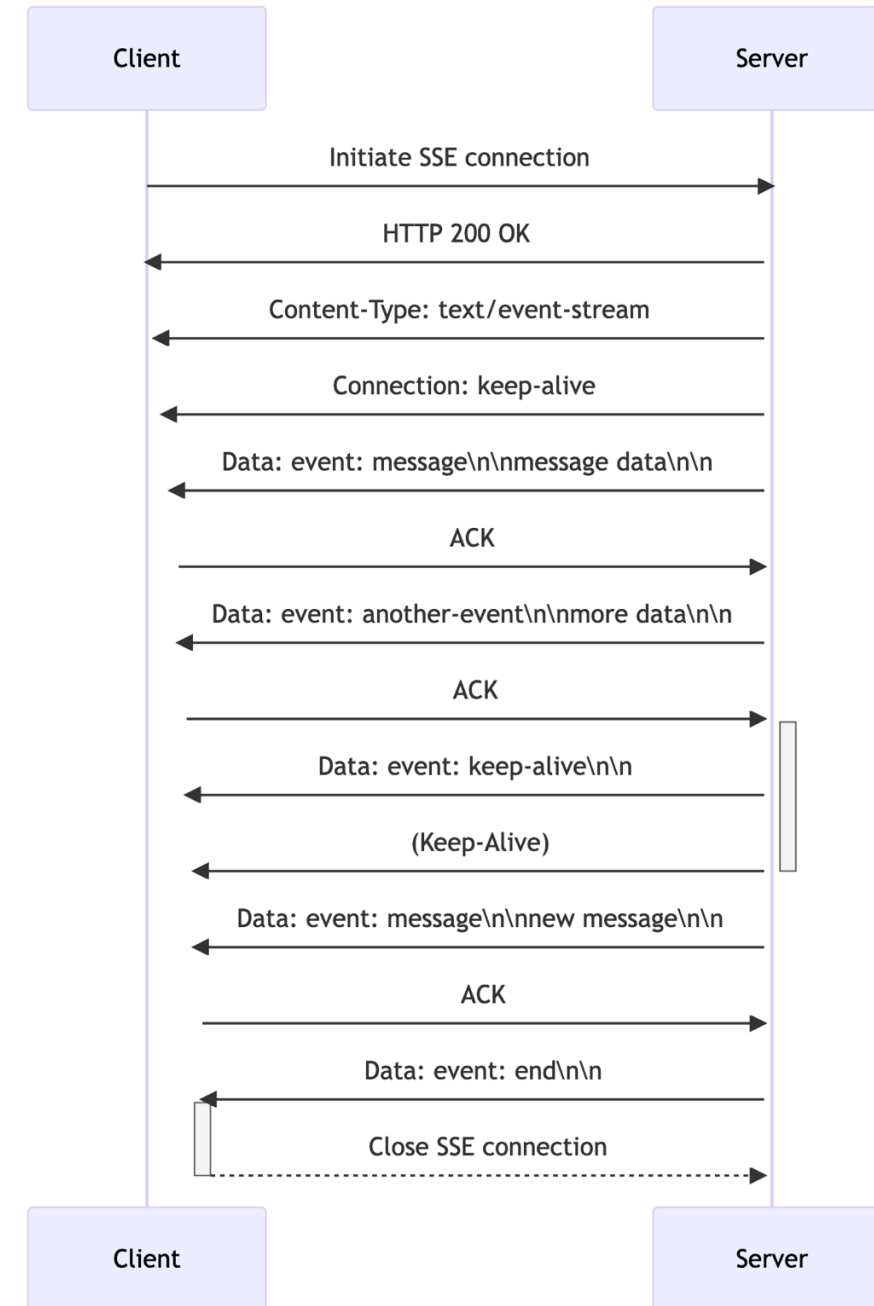


Soluções Históricas (cont.)

Server-Sent Events (SSE): Modelo de comunicação *unidirecional* em que as atualizações vêm do servidor ([server push](#)) para o cliente por meio de uma única conexão longa.

- Cliente abre uma conexão **GET /events** com cabeçalho **Accept: text/event-stream**.
- Fluxos contínuos sem necessidade de reconexão manual.
- Servidor mantém a conexão aberta e envia blocos delimitados por **\n\n**

Na imagem: diagrama de sequência de SSE. Fonte: [Code Magazine](#)



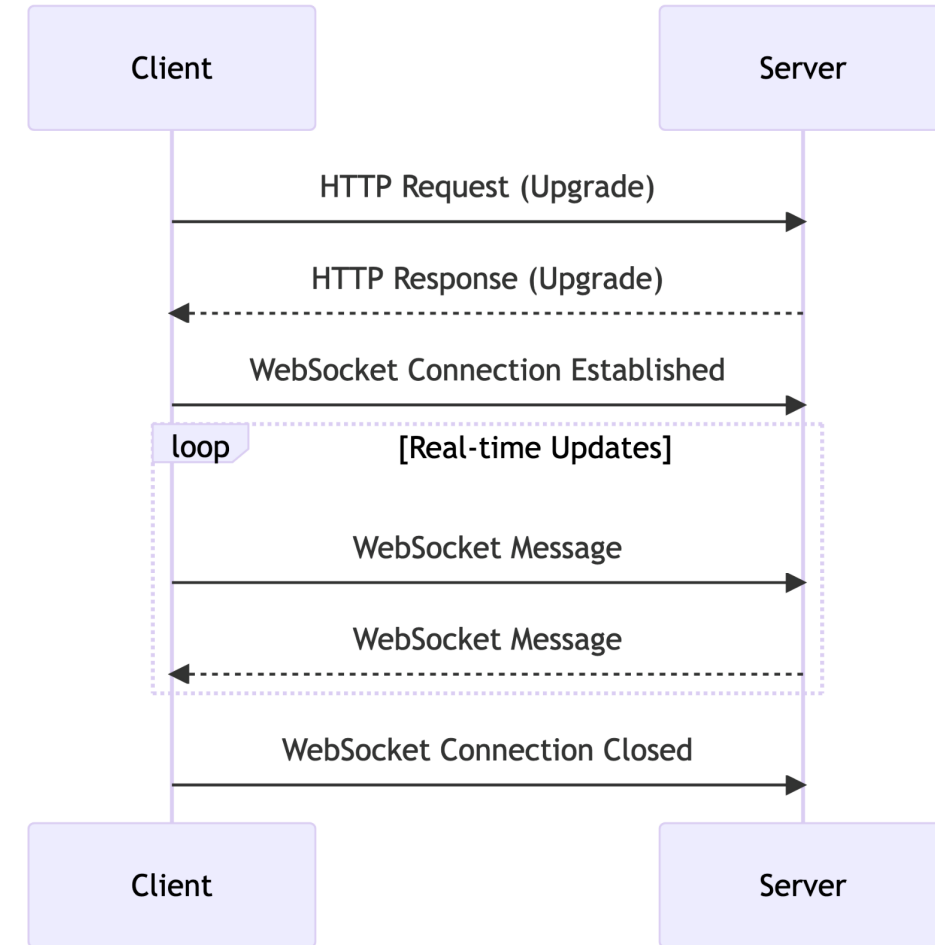
O que é WebSocket?

O protocolo [WebSocket](#), descrito na especificação [RFC 6455](#), oferece uma forma de trocar dados entre o navegador e o servidor por meio de uma conexão persistente.

- Os dados podem ser transmitidos em ambas as direções como “pacotes”, sem quebrar a conexão nem precisar de requisições HTTP adicionais.
- Permite troca de dados em tempo real com baixa latência.

Na imagem: diagrama de sequência de WebSocket.

Fonte: [Code Magazine](#).



Casos de Uso

- Chat em tempo real (ex.: Slack, Discord).
- Jogos multiplayer online.
- Dashboards com atualizações ao vivo (monitoramento).
- Colaboração em documentos (Google Docs).
- IoT: dispositivos enviando dados para a nuvem.

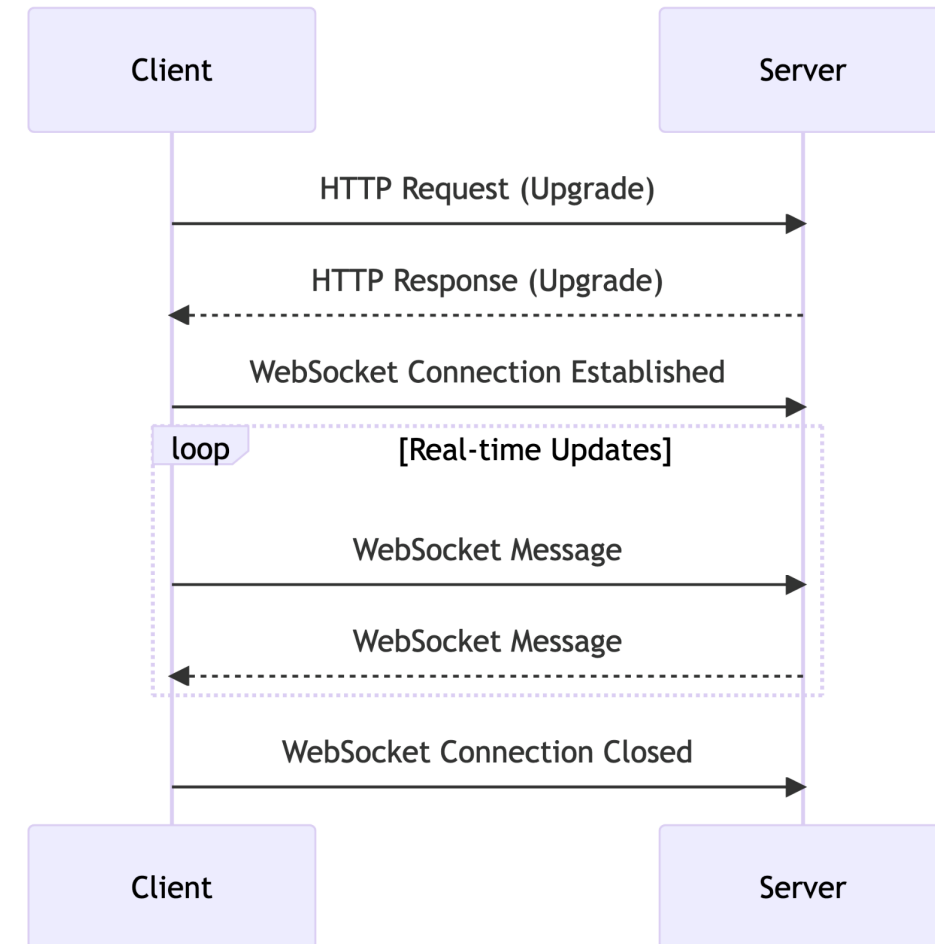
Como Funciona

Handshake: do HTTP ao WebSocket

1. **Cliente** envia **GET /chat HTTP/1.1** + **Upgrade: websocket**.
2. **Servidor** responde **101 Switching Protocols** [↗](#).
3. A partir daí a conexão permanece aberta e passa a usar frames WebSocket.

Na imagem: diagrama de sequência de WebSocket.

Fonte: [Code Magazine](#) [↗](#).



Exemplo

Request:

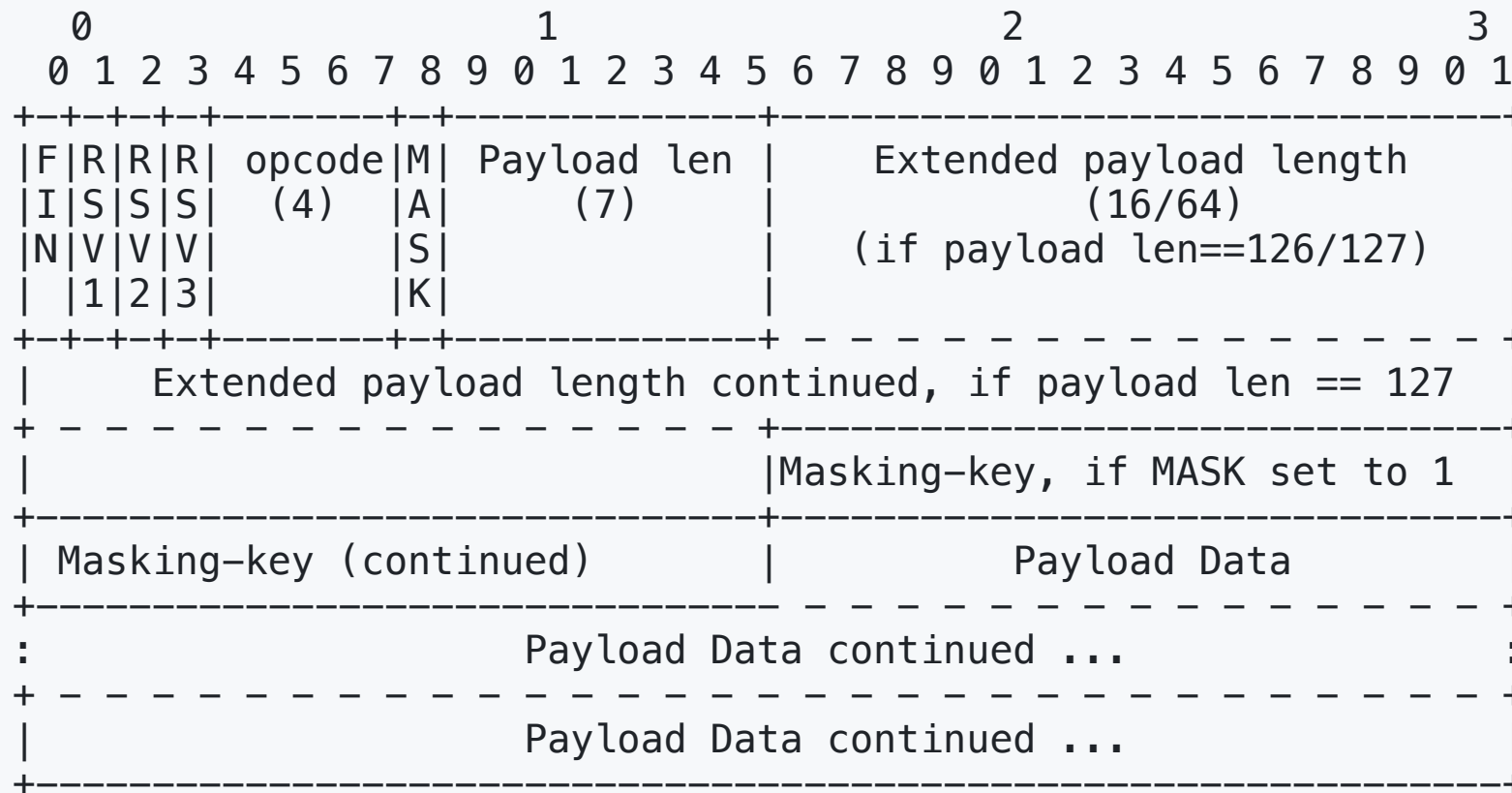
```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+x0o=
Sec-WebSocket-Protocol: chat
```

WebSocket Framing

- **Frame**: Unidade básica de dados enviada em uma conexão WebSocket.
- **Mensagem**: Um frame (não fragmentada) ou pelo menos dois frames (se fragmentada).



WebSocket Framing

Campo	Tamanho	Descrição
FIN	1 bit	Indica se este é o último frame da mensagem (1) ou não (0).
RSV1-3	3 bits	Reservados (normalmente 0).
OPCODE	4 bits	Tipo de payload (veja abaixo)
MASK	1 bit	Se o cliente enviou (1) ou não (0).
Payload length	7/16/64 bits	Tamanho do payload (até $2^{16}-1$ bytes, depois extensão).
Masking key	0/4 bytes	Presente quando MASK=1 (clientes sempre mascaram).
Payload data	N bytes	Dados da mensagem.

OPCODE: **0x1** → texto UTF-8, **0x2** → binário, **0x8** → close, **0x9** → ping, **0xA** → pong

Masking: Evita que dados binários sejam interpretados como cabeçalhos HTTP por proxies intermediários.

WebSocket vs REST

Característica	REST (HTTP)	WebSocket
Modelo de comunicação	<i>Request-Response</i> (unidirecional)	Bidirecional, fluxo contínuo
Persistência da conexão	Cada requisição abre/fecha a conexão	Conexão aberta indefinidamente
Formato de dados	JSON/XML/HTML em payload HTTP	Frames (text ou binary) sem cabeçalhos HTTP
Latência	Alta (cada chamada envolve handshake, cabeçalhos)	Baixa (apenas o frame é enviado)
Segurança	HTTPS (TLS) + autenticação padrão	wss:// (TLS); requer controle de origem e tokens
Custo de rede	Alto em chamadas frequentes	Reduzido após handshake inicial
Casos de uso típicos	CRUD, páginas estáticas, APIs públicas	Chat, jogos multiplayer, dashboards em tempo real, notificações push

A API de WebSockets no Navegador

O acesso ao protocolo no frontend é feito através da classe **WebSocket**.

1. **Setup (Conexão):** Definimos o objeto: `let socket = new WebSocket("wss://...");`
2. **Eventos (Reações):** Monitoramos os eventos para saber quando agir (*event handlers*).
3. **Métodos (Ações):** Usamos métodos específicos no objeto `socket` para enviar ou fechar o canal.

A propriedade **readyState** reflete o estado da conexão: 0 (CONNECTING), 1 (OPEN), 2 (CLOSING), 3 (CLOSED).

Exemplo Prático: JavaScript

```
let socket = new WebSocket("wss://javascript.info/article/websocket/demo/hello");

socket.onopen = function(e) {
  console.log("[open] Connection established");
  socket.send("My name is John");
};

socket.onmessage = function(event) {
  console.log(`[message] Data received from server: ${event.data}`);
};

socket.onclose = function(event) {
  if (event.wasClean) {
    console.log(`[close] Connection closed cleanly, code=${event.code} reason=${event.reason}`);
  } else {
    console.log('[close] Connection died'); // em caso de problema
  }
};

socket.onerror = function(error) {
  console.log(`[error]`);
};
```

Lab Prático

Implementar um canal de comunicação persistente, simulando um servidor que recebe mensagens e as ecoa de volta para o cliente.

Siga o passo-a-passo em:

<https://denmartins.github.io/labs/2026-04-23-sd-websockets>

Créditos da imagem: <https://unsplash.com/@safarslife>



URI

WebSocket define dois esquemas principais para a comunicação:

- **ws://**: Usado para conexões não criptografadas (equivalente ao HTTP). Mais simples, mas inseguro.
- **wss://**: Usado para conexões seguras e criptografadas via TLS/SSL (equivalente ao HTTPS). Sempre preferível em produção.

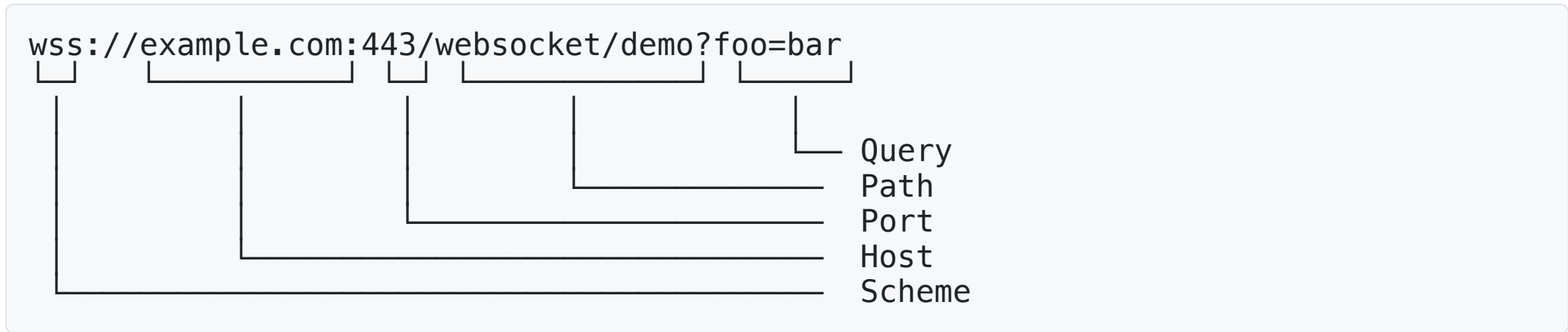
A sintaxe segue um padrão genérico, muito parecido com a URL do HTTP:

```
[Esquema]://[Host]:[Porta]/[Caminho]? [Parâmetros de Consulta]
```

Porta Padrão: A porta é opcional.

ws: Porta 80 (padrão HTTP). **wss**: Porta 443 (padrão HTTPS/TLS).

Estrutura da URI (Diagrama Conceitual)




- **Esquema** (**ws** / **wss**): Define o protocolo e a segurança.
- **Host**: O domínio do servidor (ex: **api.empresa.com**).
- **Porta**: O número de porta (Ex: 443 para **wss**, geralmente omitido).
- **Caminho** (**Path**): A rota específica no servidor (ex: **/chat**).
- **Parâmetros de Consulta** (**Query**): Informações adicionais, como filtros ou IDs (ex: **?user=alice**).

Considerações de Segurança e Travessia de Proxies

- O uso de **WSS (WebSocket Secure)** sobre TLS é essencial para garantir a confidencialidade e integridade dos dados.
- Conexões seguras (WSS) têm maior taxa de sucesso ao atravessar **proxies transparentes** e firewalls corporativos.
- A criptografia impede que intermediários analisem ou modifiquem o tráfego, tratando-o como um túnel TCP opaco.

Latência e Escalabilidade Horizontal

- WebSockets reduzem a latência ao eliminar o handshake TCP/HTTP repetitivo e o overhead de cabeçalhos.
- **Desafio de Escala:** Como conexões WebSocket são **persistentes e stateful**, o servidor deve manter o estado de cada cliente em memória.
- Para escalar, utiliza-se um **Broker de Mensagens** (Pub/Sub) para sincronizar dados entre múltiplos nós do servidor.
- O **Load Balancing** deve ser capaz de lidar com sessões persistentes e, idealmente, ser elástico para picos de tráfego.

Mais informações em <https://docs.cloud.google.com/pubsub/docs/streaming-cloud-pub-sub-messages-over-websockets?hl=pt-br> 

Heartbeats: Pings e Pongs

- Conexões inativas podem ser encerradas por roteadores ou firewalls para liberar recursos.
- O protocolo define frames de **Ping e Pong** como um mecanismo de "heartbeat".
- O servidor envia um Ping e o cliente deve responder prontamente com um Pong para provar que a conexão ainda está viva.
- Esse mecanismo permite detectar desconexões mesmo quando o soquete TCP não é fechado corretamente.

Mais informações em <https://websocket.org/guides/heartbeat/> 

Conclusão

Conceitos Principais:

- **WebSocket**: conexão persistente → latência mínima e comunicação bidirecional.
- Elimina a necessidade de *polling* constante e cabeçalhos HTTP repetitivos.
- A conexão inicia-se via **Handshake HTTP**, evoluindo para uma troca de mensagens baseada em **frames binários ou de texto** após o status **101 Switching Protocols**.

Leitura Recomendada: [The WebSocket Handbook](#) ↗

The WebSocket Handbook

Learn about the technology underpinning the realtime web and build your first web app powered by WebSockets



Dúvidas e Discussão

Exercício e Questões

Exercício Prático

Implementar uma aplicação de chat em tempo real que suporte múltiplos clientes simultâneos. O servidor deve atuar como um hub central, recebendo mensagens de qualquer cliente e retransmitindo (broadcast) essa mensagem para todos os outros clientes conectados, mantendo o canal de comunicação ativo mesmo com desconexões.

O chat deve incluir:

- Nome do usuário
- Histórico de mensagens

Questões para estudo

- Explique, com suas próprias palavras, como funciona o modelo de comunicação do protocolo HTTP tradicional e por que ele não é ideal para aplicações em tempo real.
- Descreva o processo de estabelecimento de uma conexão WebSocket desde o momento em que o cliente inicia a comunicação até a conexão ser estabelecida.
- Compare o uso de WebSockets e APIs REST em aplicações web modernas. Em quais cenários cada um deve ser utilizado?
- Discuta os principais desafios de escalabilidade ao utilizar WebSockets em aplicações com muitos usuários simultâneos.
- Suponha que você precisa desenvolver um sistema de monitoramento em tempo real de sensores industriais. Justifique tecnicamente por que WebSockets seriam mais adequados que HTTP.