Long Polling vs WebSockets

Ashish Pratap Singh

5 min read

Whether you are playing an online game or chatting with a friend—updates appear in real-time without hitting **"refresh"**.

Behind these seamless experiences lies a critical engineering decision: **how to push real-time updates from servers to clients**.

The traditional HTTP model was designed for request-response: *"Client asks, server answers."*. But in many real-time systems, the server needs to talk first and more often.

This is where **Long Polling** and **WebSockets** come into play—two popular methods for achieving real-time updates.

In this chapter, we'll explore these two techniques, how they work, their pros and cons, and use cases.

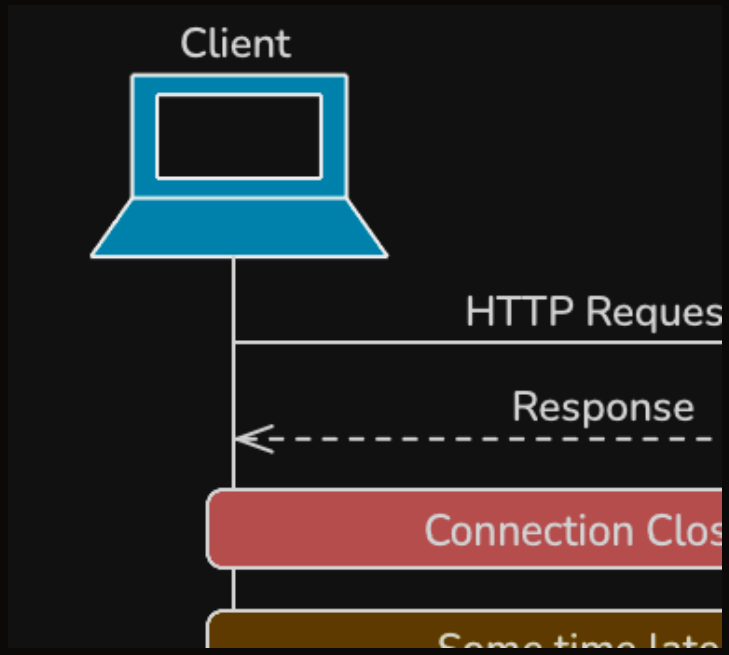
Reading Progress 0%

On this page

1. Why Traditional HTTP Isn't Enough

HTTP, the backbone of the web, follows a **client-driven request-response model**:

1. The client (e.g., a browser or mobile app) sends a request to the server.
2. The server processes the request and sends back a response.
3. The connection closes.



This model is simple and works for many use-cases, but it

has limitations:

- **No automatic updates:** With plain HTTP, the server cannot proactively push data to the client. The client has to request the data periodically.
- **Stateless nature:** HTTP is stateless, meaning each request stands alone with no persistent connection to the server. This can be problematic if you need continuous exchange of data.

To build truly real-time features—live chat, financial tickers, or gaming updates—you need a mechanism where the server can instantly notify the client when something changes.

---

2. Long Polling

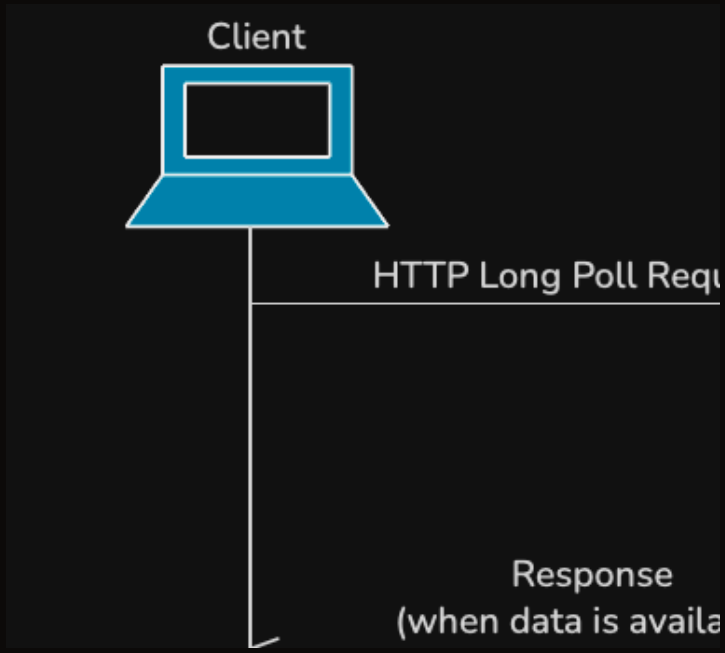
Long polling is a technique that mimics real-time behavior by keeping HTTP requests open until the server has data.

Long Polling is an enhancement over traditional polling. In regular polling, the client repeatedly sends requests at fixed intervals (e.g., every second) to check for updates. This can be wasteful if no new data exists.

Long Polling tweaks this approach: the client asks the

server for data and then “waits” until the server has something new to return or until a timeout occurs.

How Does Long Polling Work?



1. **Client sends a request** to the server, expecting new data.
2. **Server holds the request open** until it has an update or a timeout is reached.
    - If there's new data, the server immediately responds.
    - If there’s no new data and the timeout is reached, the server responds with an empty or minimal message.

3. Once the client receives a response—new data or a timeout—it **immediately sends a new request** to the server to keep the connection loop going.

Pros ✅

- Simple to implement (uses standard HTTP).
- Supported universally since it uses standard HTTP, and it works reliably through firewalls and proxies.

Cons ❌

- Higher latency after each update (client must re-establish connection).
- Resource-heavy on servers (many open hanging requests).

Use Cases

- Simple chat or comment systems where real-time but slightly delayed updates (near real-time) are acceptable.
- Notification systems for less frequent updates (e.g., Gmail's "new email" alert).
- Legacy systems where WebSockets aren't feasible.

Code Example (JavaScript)

Javascript

```javascript
// Client-side long polling
async function fetchUpdates() {
  try {
    const response = await fetch('/updates');
    const data = await response.json();
    console.log('Update:', data);
    fetchUpdates(); // Re-run immediately
  } catch (error) {
    setTimeout(fetchUpdates, 5000); // Retry after 5
  }
}

fetchUpdates();
```

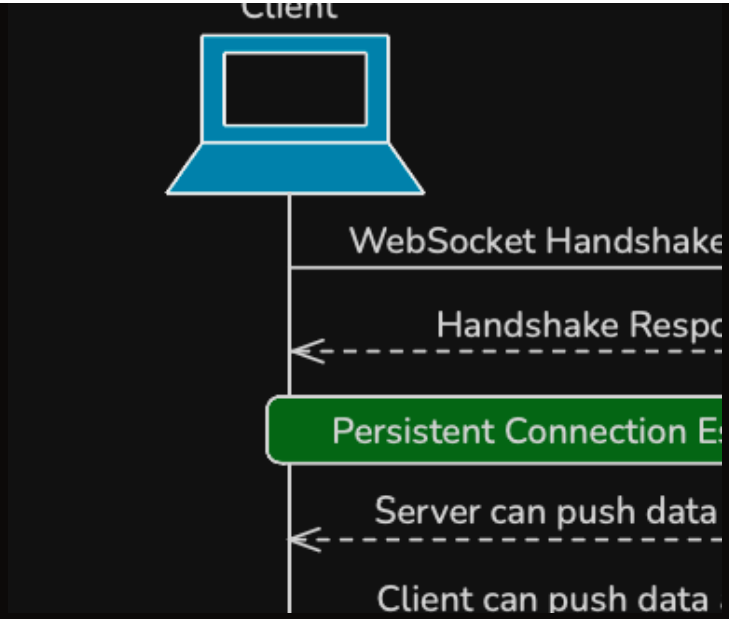
---

3. WebSockets

WebSockets provide a **full-duplex, persistent connection** between the client and the server.

Once established, both parties can send data to each other at any time, without the overhead of repeated HTTP requests.

How Do WebSockets Work?



1. **Handshake:** Client sends an HTTP request with `Upgrade: websocket`.
2. **Connection**: If supported, the server upgrades the connection to WebSocket (switching from `http://` to `ws://`). After the handshake, client and server keep a TCP socket open for communication.
3. **Full-Duplex Communication:** Once upgraded, data can be exchanged bidirectionally in real time until either side closes the connection.

Pros ✅

1. Ultra-low latency (no repeated handshakes).
2. Lower overhead since there's only one persistent con-

nection rather than repeated HTTP requests.
3. Scalable for real-time applications that need to support large number of concurrent users.

Cons ❌

1. More complex setup (requires the client and server to support WebSocket).
2. Some proxies and firewalls may not allow WebSocket traffic.
3. Complexity in implementation and handling reconnections/errors.
4. Server resource usage might grow if you have a large number of concurrent connections.

Use Cases

1. Live chat and collaboration tools (Slack, Google Docs, etc.).
2. Multiplayer online games with real-time state synchronization.
3. Live sports/financial dashboards that need to push frequent updates.

Code Example (JavaScript)

Javascript

```javascript
// Client-side WebSocket
const socket = new WebSocket('wss://yourserver.com/v

socket.onopen = () => {
  socket.send('Hello Server!');
};

socket.onmessage = (event) => {
  console.log('Update:', event.data);
};

// Server-side (Node.js with ws library)
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws) => {
  ws.on('message', (message) => {
    ws.send(`Server received: ${message}`);
  });
});
```

---

4. Choosing the Right Solution

Both methods achieve real-time updates, but your choice depends on your project's requirements:

1. **Complexity and Support**
    - **Long Polling** is easier to implement using standard libraries. Any environment that supports HTTP can handle it, often without extra packages.
    - **WebSockets** require a bit more setup and a capable proxy environment (e.g., support in Nginx or HAProxy). However, many frameworks (e.g., Socket.io) simplify the process significantly.

2. **Scalability and Performance**
    - **Long Polling** can become resource-intensive with a large number of simultaneous clients, due to multiple open connections waiting on the server side.
    - **WebSockets** offer a more efficient, persistent connection and scale better for heavy, frequent data streams.

3. **Type of Interaction**
    - **Long Polling** fits scenarios where data updates aren't super frequent. If new data arrives every few seconds or minutes, long polling might be enough.
    - **WebSockets** are better for high-frequency updates or two-way communication (e.g., multiple participants editing a document or interacting in a game).

4. **Network Constraints**

- **Long Polling** typically works even in older networks or those with strict firewalls.
- **WebSockets** might face issues in certain corporate or older mobile environments, though this is less of a problem as the standard becomes more widespread.

> While both achieve real-time communication, WebSockets are generally more efficient for truly real-time applications, while Long Polling can be simpler to implement for less demanding scenarios.

---

5. Alternative Solutions Worth Considering

1. Server-Sent Events (SSE)

- Allows the server to push messages to the client over HTTP.
- It's simpler than WebSockets for one-way communication, but not full-duplex.
- Best suited for use cases like news feeds, real-time notifications, and status updates.

2. MQTT

- Commonly used in IoT for lightweight publish-subscribe messaging.
- Specialized for device-to-device or device-to-server communication with minimal overhead.

3. Libraries like Socket.io

- Provides an abstraction over WebSockets for easier real-time communication.
- Automatically falls back to long polling if WebSockets are unsupported.
- Ensures cross-browser compatibility with robust and reliable performance.