**Long Polling vs WebSockets**

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 article, we’ll explore these two techniques, how they work, their pros and cons, and use cases.

**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.

[[A diagram of a computer

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!2MXM!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F094ef5bb-a62f-44b4-8ef0-a317c3504e61_410x558.png)](https://substackcdn.com/image/fetch/$s_!2MXM!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F094ef5bb-a62f-44b4-8ef0-a317c3504e61_410x558.png" \t "_blank)

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 (সার্ভারের সাথে কোন স্থায়ী সংযোগ নেই). 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 (অনুকরণ করে)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.

[[A diagram of a process flow

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!OZ48!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Ff1a03057-991b-4334-80a2-5d413f7d4a61_449x620.png)](https://substackcdn.com/image/fetch/$s_!OZ48!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Ff1a03057-991b-4334-80a2-5d413f7d4a61_449x620.png" \t "_blank)

**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)**

[[A computer screen shot of text

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!ps4_!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6e16c3c7-3e02-4c48-9db1-2de7c35c1689_595x339.png)](https://substackcdn.com/image/fetch/$s_!ps4_!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6e16c3c7-3e02-4c48-9db1-2de7c35c1689_595x339.png" \t "_blank)

**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.

[[A diagram of a computer

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!MzeY!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F89167db8-498e-4c23-bd68-c61935128b27_410x572.png)](https://substackcdn.com/image/fetch/$s_!MzeY!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F89167db8-498e-4c23-bd68-c61935128b27_410x572.png" \t "_blank)

**Pros ✅**

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

[[A computer screen shot of a program code

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!Tw27!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F2692cd8b-09c4-4c76-8852-fddbe0549399_557x521.png)](https://substackcdn.com/image/fetch/$s_!Tw27!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F2692cd8b-09c4-4c76-8852-fddbe0549399_557x521.png" \t "_blank)

**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.

Thank you for reading!

**1. Polling**

* **How it works:**  
  Client sends requests to the server at regular intervals (e.g., every 5 seconds) to check if there’s new data.
* **Pros:**
  + Simple to implement.
  + Works with any HTTP server.
* **Cons:**
  + Wastes resources (many empty requests).
  + Not real-time, only as fast as polling interval.

👉 **Example use case:** Stock price updates every 10s, live scores (not critical real-time).

**🔹 2. Long Polling**

* **How it works:**
  + Client sends a request to the server.
  + If the server has no data yet, it holds the connection open until new data is available (or timeout).
  + When the server responds, client immediately sends another request.
* **Pros:**
  + Simulates real-time over HTTP.
  + Works with older infrastructure.
* **Cons:**
  + Still many HTTP connections.
  + More server-side complexity.

👉 **Example use case:** Chat applications before WebSockets.

**🔹 3. WebSocket**

* **How it works:**
  + Client and server establish a **persistent, bidirectional connection** (via HTTP Upgrade).
  + Once connected, either side can send messages anytime.
* **Pros:**
  + True **real-time** communication.
  + Very efficient (no repeated HTTP overhead).
  + Supports duplex communication.
* **Cons:**
  + Requires WebSocket support in server + infra.
  + More complex than HTTP polling.

👉 **Example use case:** Multiplayer games, collaborative editing (Google Docs), real-time dashboards.

**🔹 4. Alternatives / Related Approaches**

Besides these three, there are some modern alternatives:

**🔸 Server-Sent Events (SSE)**

* One-way stream: server → client.
* Uses plain HTTP (no special protocol).
* Great for real-time feeds like notifications, news updates.
* **Limitation:** Client can’t send data back over SSE (needs separate request).

**🔸 gRPC with Streaming**

* Uses **HTTP/2** for streaming requests/responses.
* Can do:
  + Client streaming
  + Server streaming
  + Bidirectional streaming
* More efficient than HTTP/1.1 approaches.

**🔸 MQTT**

* Lightweight pub/sub protocol, very efficient over low bandwidth.
* Used a lot in **IoT** and mobile apps.

**🔸 WebTransport / WebRTC (newer)**

* WebRTC: peer-to-peer communication (video/audio/data channels).
* WebTransport: replacement for WebSocket over QUIC, supports multiplexed streams.

**✅ Quick Comparison**

| **Feature** | **Polling** | **Long Polling** | **WebSocket** | **SSE** | **gRPC Streaming** | **MQTT** |
| --- | --- | --- | --- | --- | --- | --- |
| Direction | Client → Server | Mostly server → client | Bidirectional | Server → Client | Bi/Uni-direction | Pub/Sub |
| Real-time efficiency | ❌ No | ⚠️ Better | ✅ Best | ✅ Good | ✅ Excellent | ✅ Excellent |
| Overhead | High | Medium | Low | Low | Low | Very Low |
| Browser Support | ✅ Full | ✅ Full | ✅ Full | ✅ Full | ❌ Limited | ❌ Limited |
| Use Case | Simple data | Chat apps (legacy) | Games, trading, chat | Notifications | Microservices | IoT |

👉 If you’re building a **web app today**:

* **Use WebSockets** for **bidirectional real-time**.
* **Use SSE** if you only need **server → client push**.
* **Use Polling/Long Polling** only as fallback for old environments.
* **Use gRPC / MQTT** if working with microservices or IoT.

**Interview Questions & Answers**

**1. What is the difference between polling, long polling, and WebSockets?**

**Answer:**

* **Polling:** Client repeatedly requests data at fixed intervals.
* **Long Polling:** Client requests data, server holds connection until new data is available.
* **WebSocket:** Persistent, bidirectional connection where server and client can push data anytime.

**2. Why is polling considered inefficient for real-time communication?**

**Answer:**  
Because it sends requests even when no new data is available, causing unnecessary **network traffic and server load**.

**3. How does long polling improve upon standard polling?**

**Answer:**  
Long polling reduces wasted requests by keeping the connection open until data is available, making it **closer to real-time** without constant polling.

**4. How does WebSocket establish a connection with the server?**

**Answer:**  
Through an **HTTP Upgrade handshake** (Switching Protocols). The connection then transitions to a persistent **TCP socket** for duplex communication.

**5. What are the advantages and disadvantages of WebSockets compared to HTTP polling?**

**Answer:**

* **Advantages:** Real-time, low latency, less overhead, bidirectional.
* **Disadvantages:** More complex setup, requires server and infra support, may need load balancing considerations.

**6. In which scenarios would you prefer SSE over WebSockets?**

**Answer:**  
When you only need **server → client one-way push**, like live news feeds, notifications, or social media updates.

**7. Can WebSockets work without an HTTP server?**

**Answer:**  
Yes, technically WebSocket is just a protocol over TCP, but in practice it usually starts via **HTTP handshake**, so servers typically support both.

**8. How do you handle scalability with WebSockets in a distributed system?**

**Answer:**  
By using **sticky sessions** or a **message broker (Redis, Kafka, RabbitMQ)** to distribute WebSocket messages across multiple servers.

**9. What are the differences between SSE and WebSocket in terms of direction and performance?**

**Answer:**

* **SSE:** One-way (server → client), lightweight, uses HTTP, auto-reconnect.
* **WebSocket:** Two-way, real-time, lower latency, but more complex.

**10. What are the key differences between gRPC streaming and WebSockets?**

**Answer:**

* **gRPC streaming:** Built on HTTP/2, supports server, client, and bidirectional streaming. Best for **microservices**.
* **WebSockets:** Built on TCP after HTTP Upgrade, best for **real-time web apps**.

**11. You are building a chat application — which technology would you choose and why?**

**Answer:**  
**WebSocket**, because it requires **bidirectional, low-latency communication** between multiple clients and the server.

**12. You need to send real-time stock price updates to thousands of users. Which approach is best?**

**Answer:**  
**WebSockets** (for full interactivity) or **SSE** (if only server → client updates).

**13. Your system must send server → client notifications only. Which technology fits best?**

**Answer:**  
**SSE** (simpler than WebSocket, auto-reconnect, lightweight).

**14. How would you implement fallback if WebSocket is not supported in some environments?**

**Answer:**  
Use a library like **Socket.IO**, which falls back to **long polling** automatically if WebSockets are unavailable.

**15. You are designing an IoT system with limited bandwidth. Would you use WebSocket or MQTT? Why?**

**Answer:**  
**MQTT**, because it is a **lightweight pub/sub protocol** optimized for unreliable networks and low bandwidth.

**16. Suppose you have a real-time dashboard showing analytics updates every second. Which communication strategy is best?**

**Answer:**  
**WebSockets** for efficient continuous updates, or **SSE** if only server push is required.

**17. How would you detect a disconnected WebSocket client and handle reconnection?**

**Answer:**

* Use **ping/pong heartbeat messages**.
* If no response within timeout, assume disconnection and attempt **automatic reconnection**.

**18. How would you handle load balancing in a WebSocket-enabled application?**

**Answer:**

* Use **sticky sessions** to bind a client to the same server.
* Or use a **pub/sub broker** (Redis/Kafka) so all servers can share WebSocket messages.

**19. Explain how HTTP/2 multiplexing impacts real-time communication compared to HTTP/1.1.**

**Answer:**  
HTTP/2 allows **multiple streams over a single connection**, reducing latency and overhead compared to HTTP/1.1, which requires one request per connection.

**20. Compare WebRTC vs WebSocket — when would you choose one over the other?**

**Answer:**

* **WebRTC:** Peer-to-peer, low-latency video, audio, and data transfer (e.g., Zoom, Google Meet).
* **WebSocket:** Client-server real-time messaging (e.g., chat, dashboards).