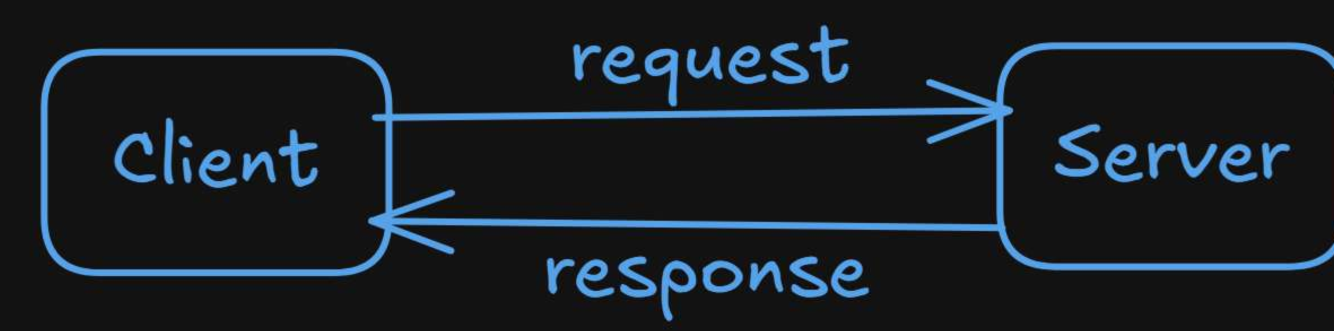
Why WebSockets Came (What was broken in HTTP)



Original HTTP Model (Request → Response)

HTTP was designed for documents, not real-time systems.

Basic rule of HTTP:
-> Client sends a request
-> Server sends a response
-> Connection closes (HTTP/1.1: mostly)

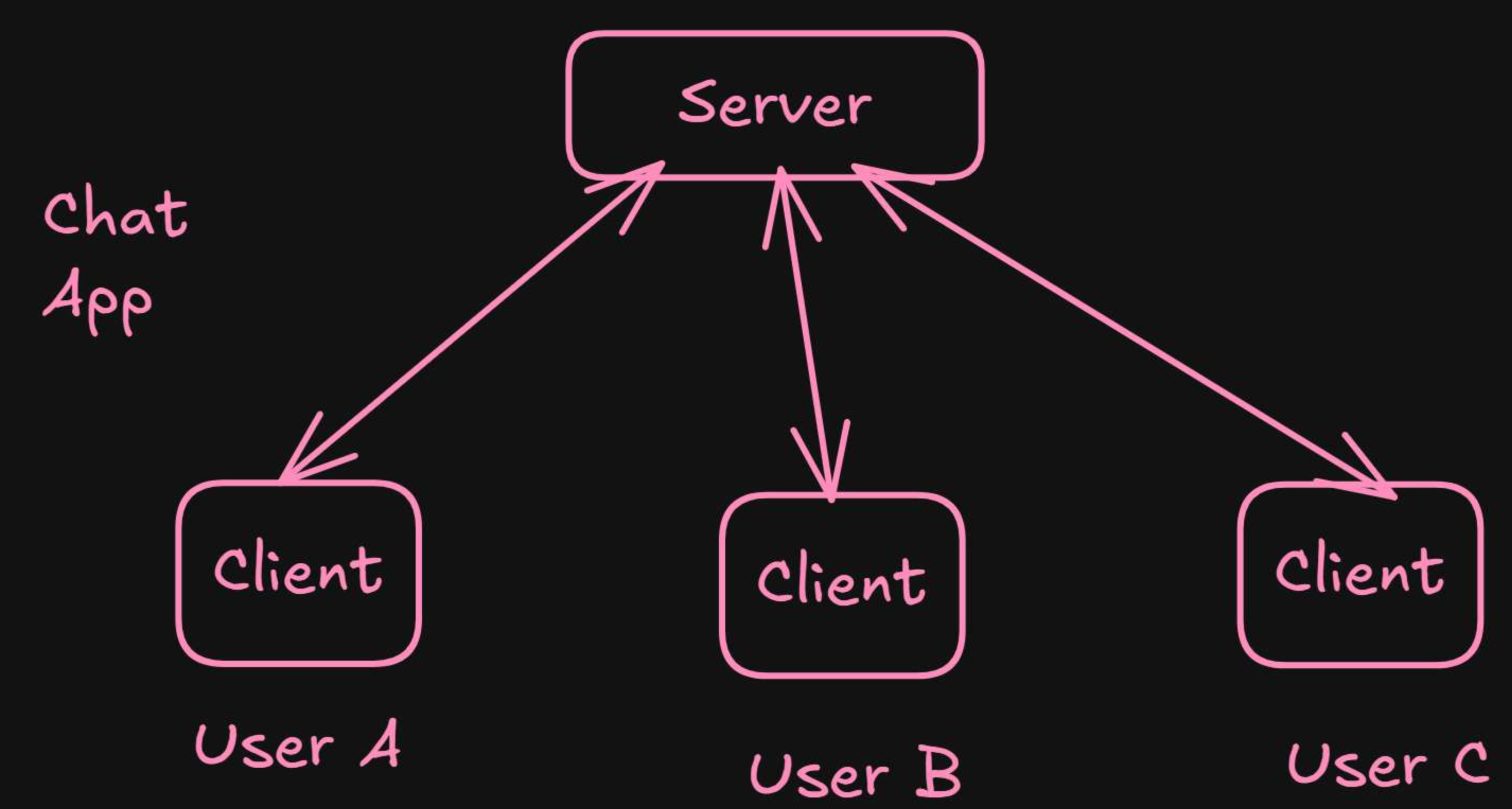
That's it.

Problem:

What if the server has new data, but the client didn't ask?

-> Server cannot push data.

Real-world Use Cases That Broke HTTP

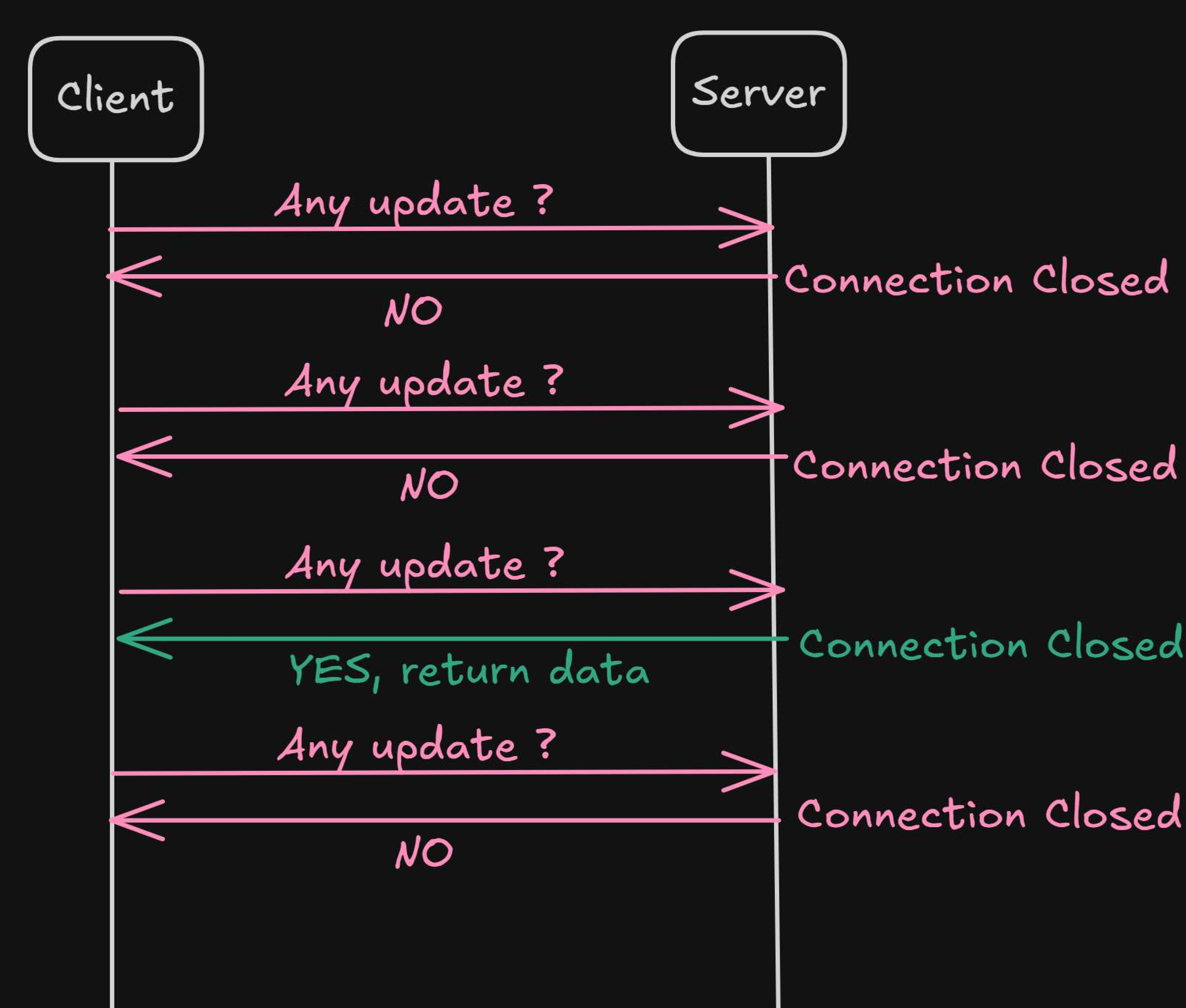


Modern apps need real-time:

Chat apps (WhatsApp, Slack)
Stock price updates
Notifications
Multiplayer games
Live dashboards
Online collaboration (Google Docs)

HTTP alone cannot do this efficiently.

Hacks Before WebSockets (How we tried to fix HTTP)



1. Polling (The worst but simplest)

Client asks server every X seconds
"Any update?"

Problems:

-> Huge number of useless requests
-> High latency (updates delayed until next poll)
-> Wasted CPU, bandwidth, money

Example:
10M users × poll every 5 sec = 120M requests/minute

2. Long Polling (Slightly better)

-> Client sends request

-> Server waits until data is available

-> Server responds

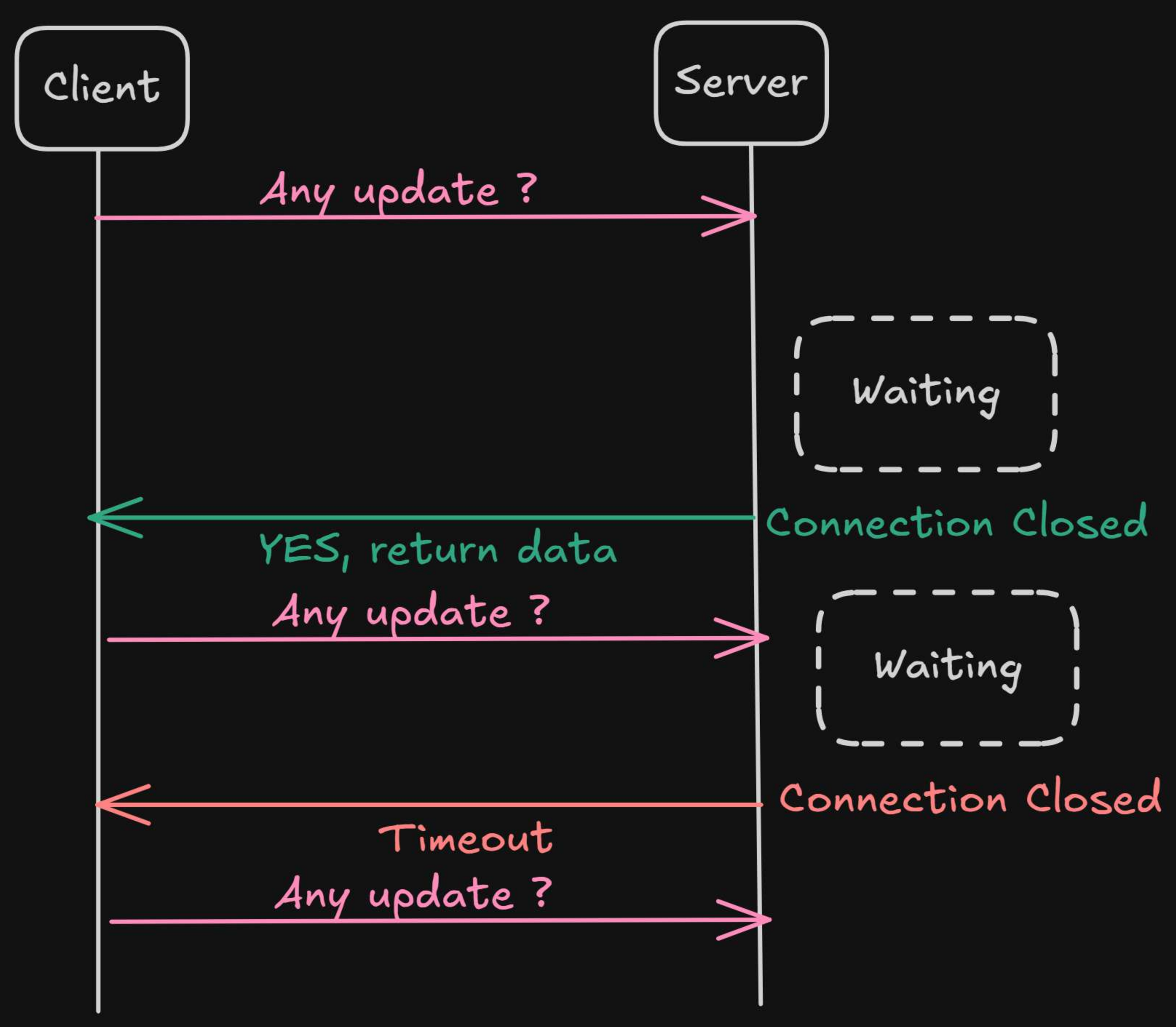
-> Client immediately sends another request

Problems:

Still HTTP overhead (headers again & again)

Connection constantly created & destroyed

Hard to scale



3. Server-Sent Events (SSE)

-> Client opens one HTTP connection

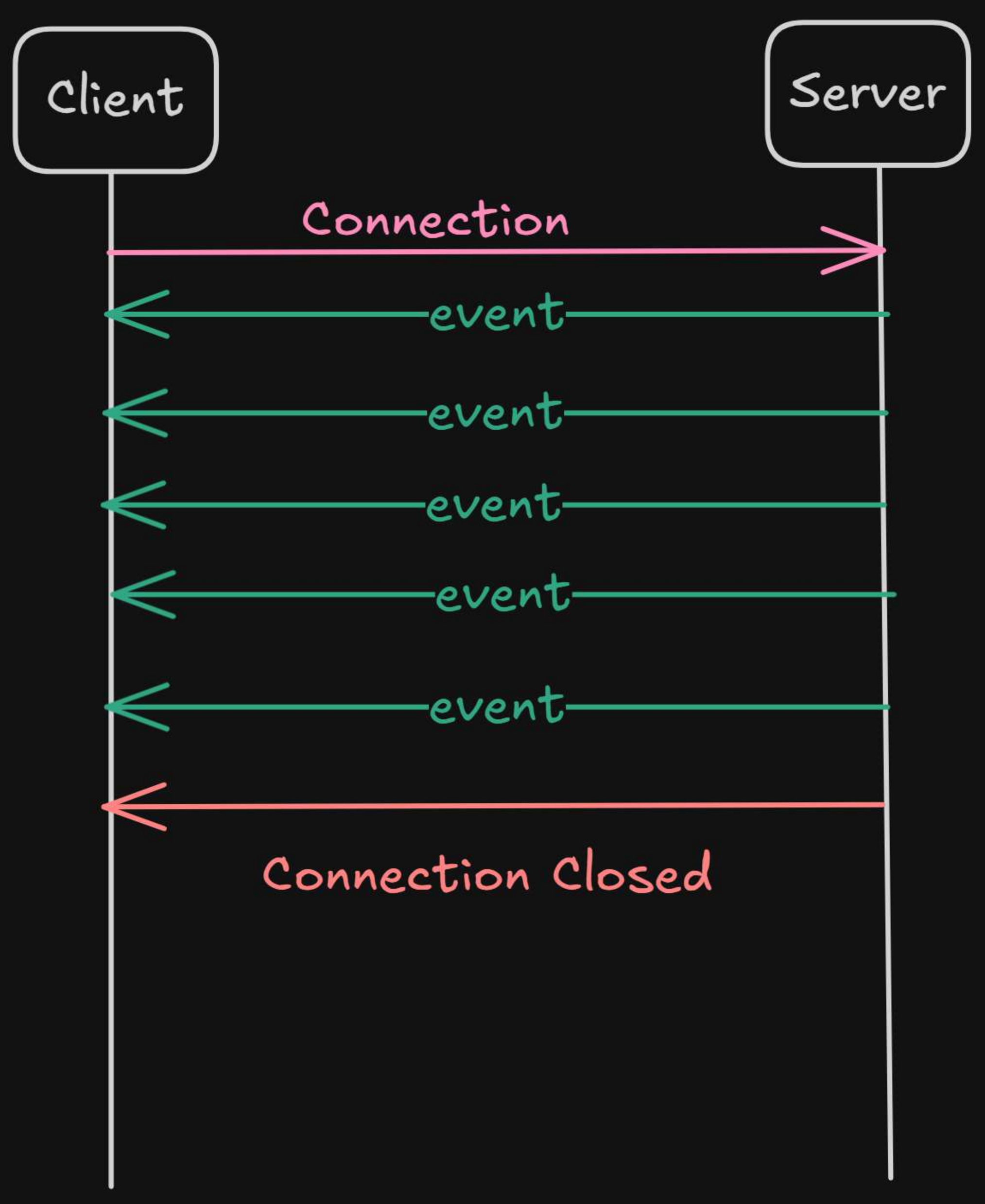
-> Server pushes updates

Limitations:

-> One-way only (server → client)

-> No binary data

-> Not suitable for chat/games



HTTP was never meant for real-time

Why WebSockets Were Invented ?

Goal of WebSockets

Create a protocol that:

-> Keeps one persistent connection

-> Allows full-duplex communication

-> Has low overhead

-> Works well with browsers

-> Plays nicely with existing HTTP infrastructure

WebSockets were standardized

What Exactly Is WebSocket?

A persistent, full-duplex communication protocol
over a single TCP connection

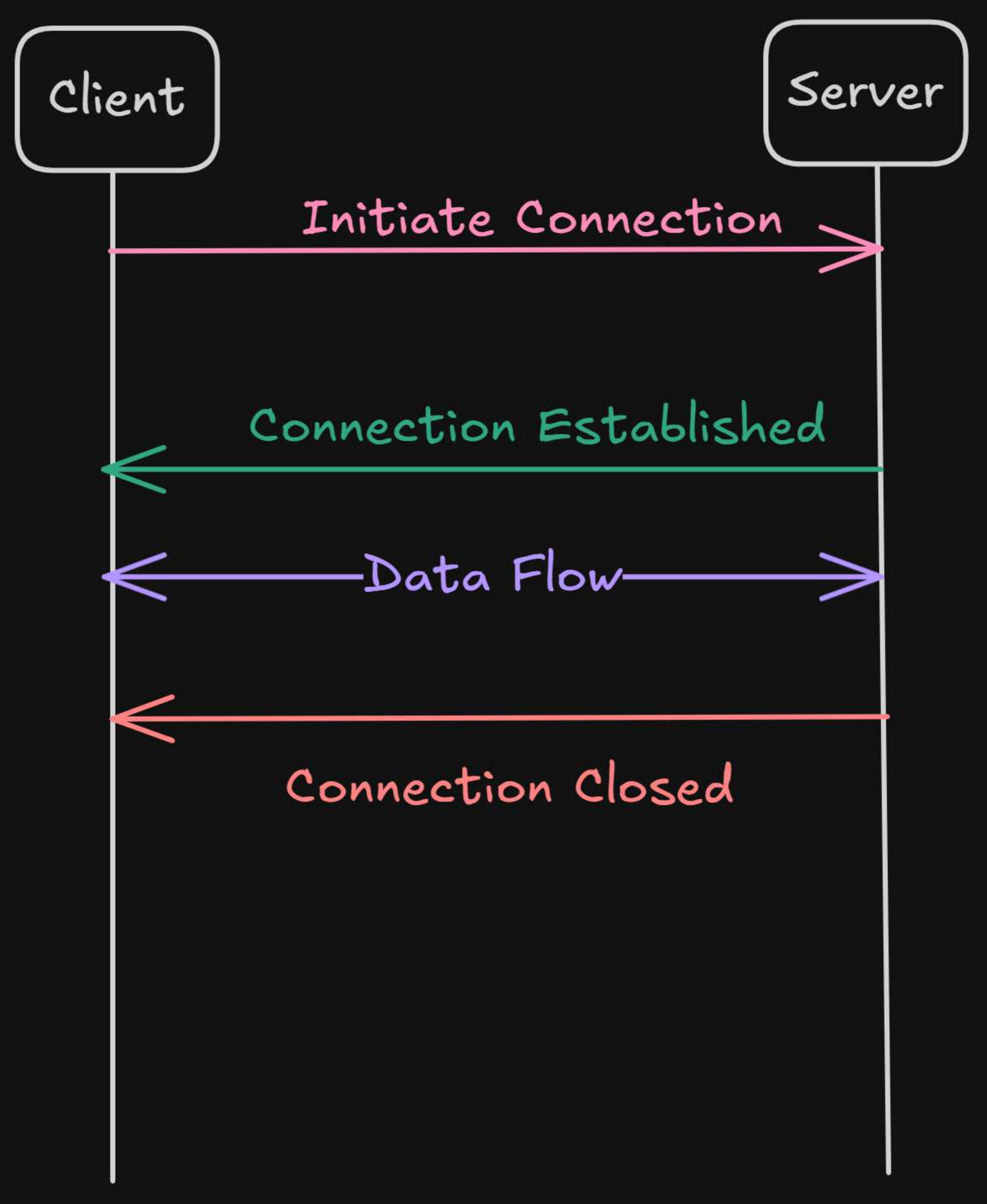
Key Properties:

-> One connection

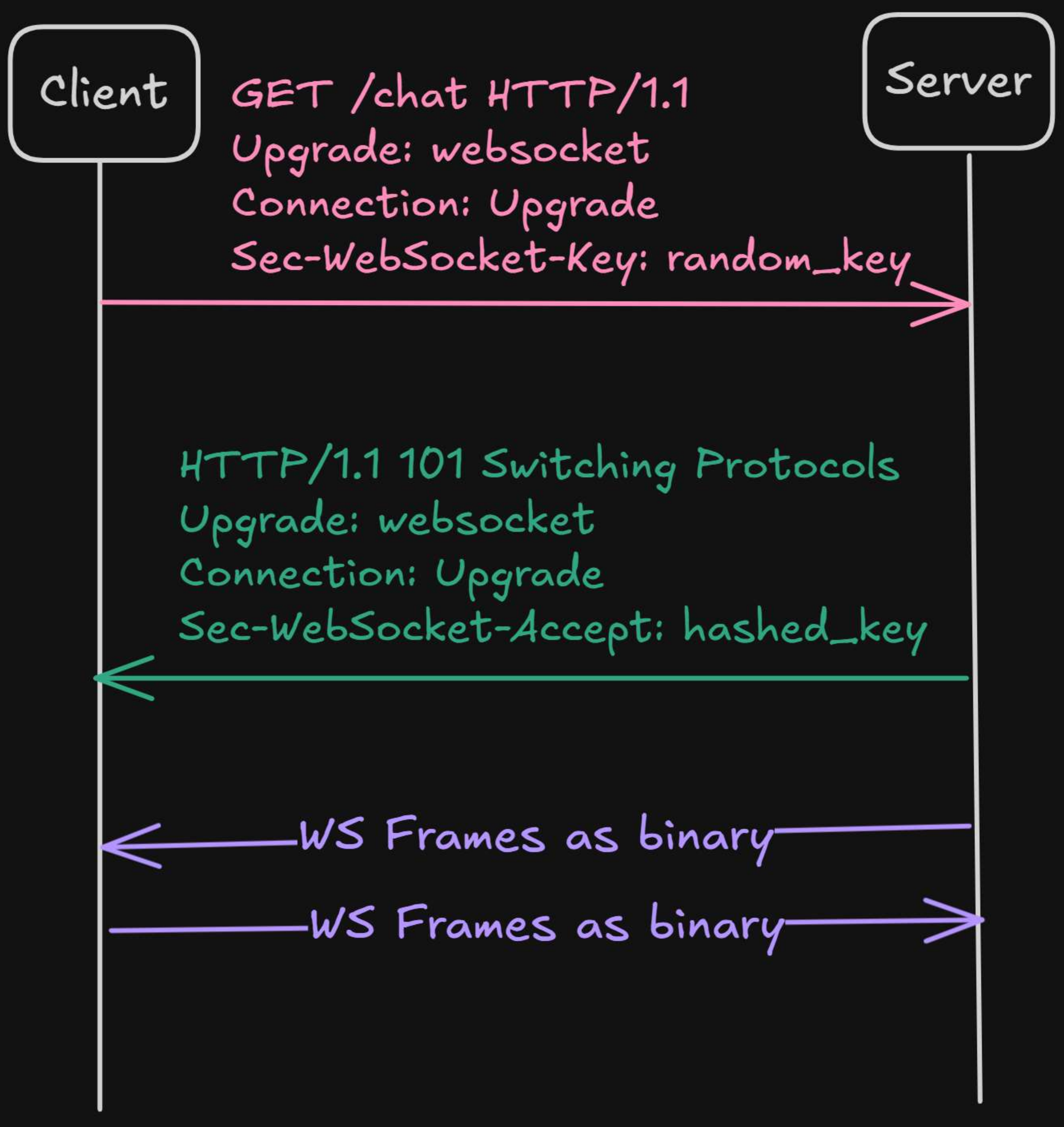
-> Client ↔ Server both can send anytime

-> Very low latency

-> Minimal overhead



How WebSocket Connection Is Established (Handshake)



Step 0 — TCP connection (happens silently)

Before anything else:

-> Client opens TCP connection to server
  -> Port 80 (ws) or 443 (wss)

-> If wss → TLS handshake happens

This is normal TCP/HTTPS, nothing special yet.

Step 1 — Client sends HTTP request (Upgrade request)

Only the CLIENT can start the upgrade

Client sends a normal HTTP request with special headers:

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: random_key
Sec-WebSocket-Version: 13
```

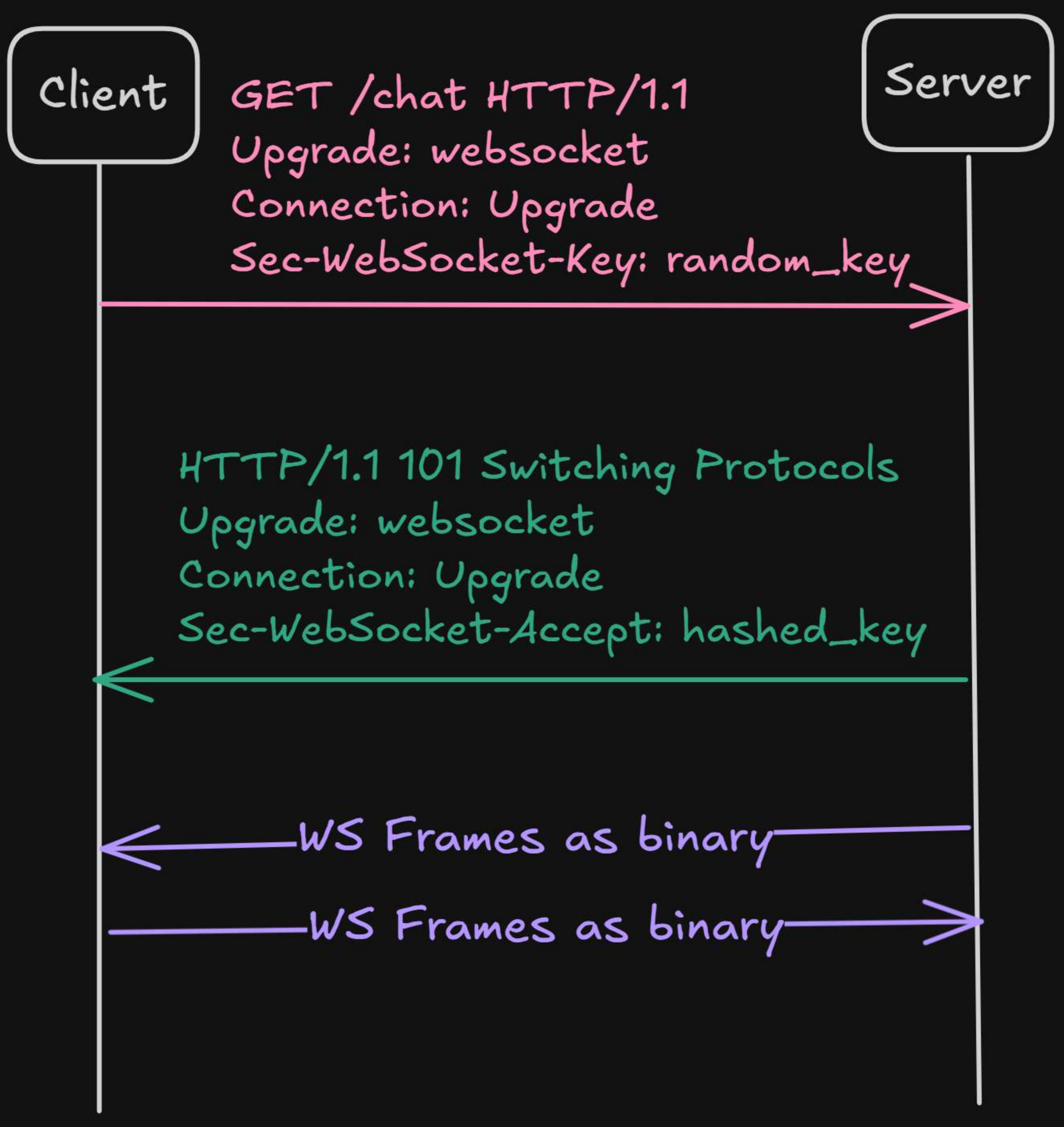
What each header means

Upgrade: websocket
-> "I want to switch protocol"

Connection: Upgrade
-> "This connection should be upgraded"

Sec-WebSocket-Key
-> Random value (security check)

Sec-WebSocket-Version: 13
-> WebSocket protocol version

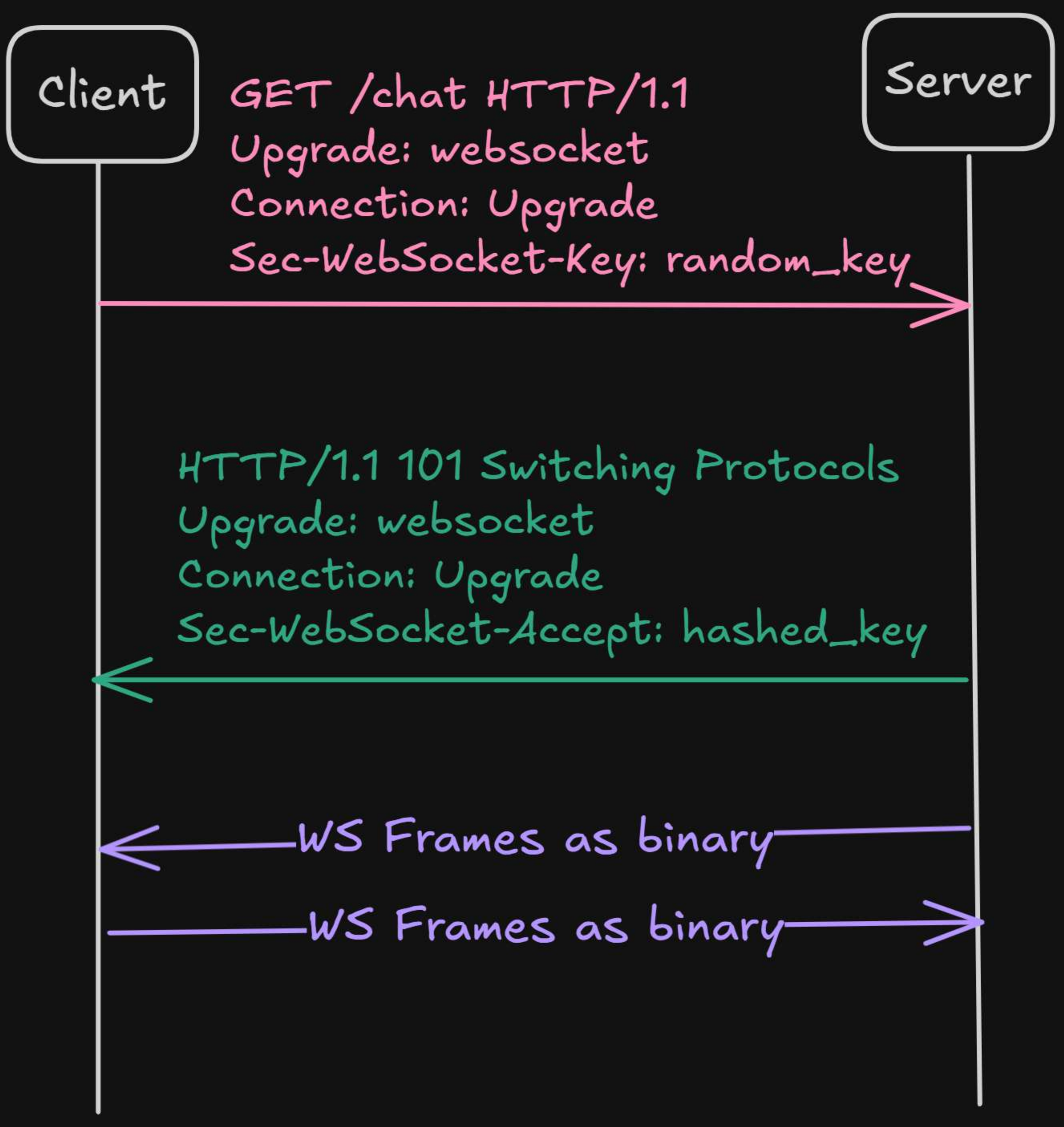


Important:

This is still pure HTTP

No WebSocket frames yet

Step 2 — Server validates the request



Server checks:

-> Is Upgrade: websocket present?
-> Is version 13 supported?
-> Is this endpoint allowed to upgrade?
-> If any check fails → server replies with normal HTTP error

Step 3 — Server sends HTTP 101 response (Upgrade accepted)

Step 4 — Protocol switches (this is the magic moment)

After 101 response:

-> No more HTTP
-> WebSocket protocol starts

From this point:

No HTTP headers
No request/response model
Only WebSocket frames

Step 5 — WebSocket communication begins

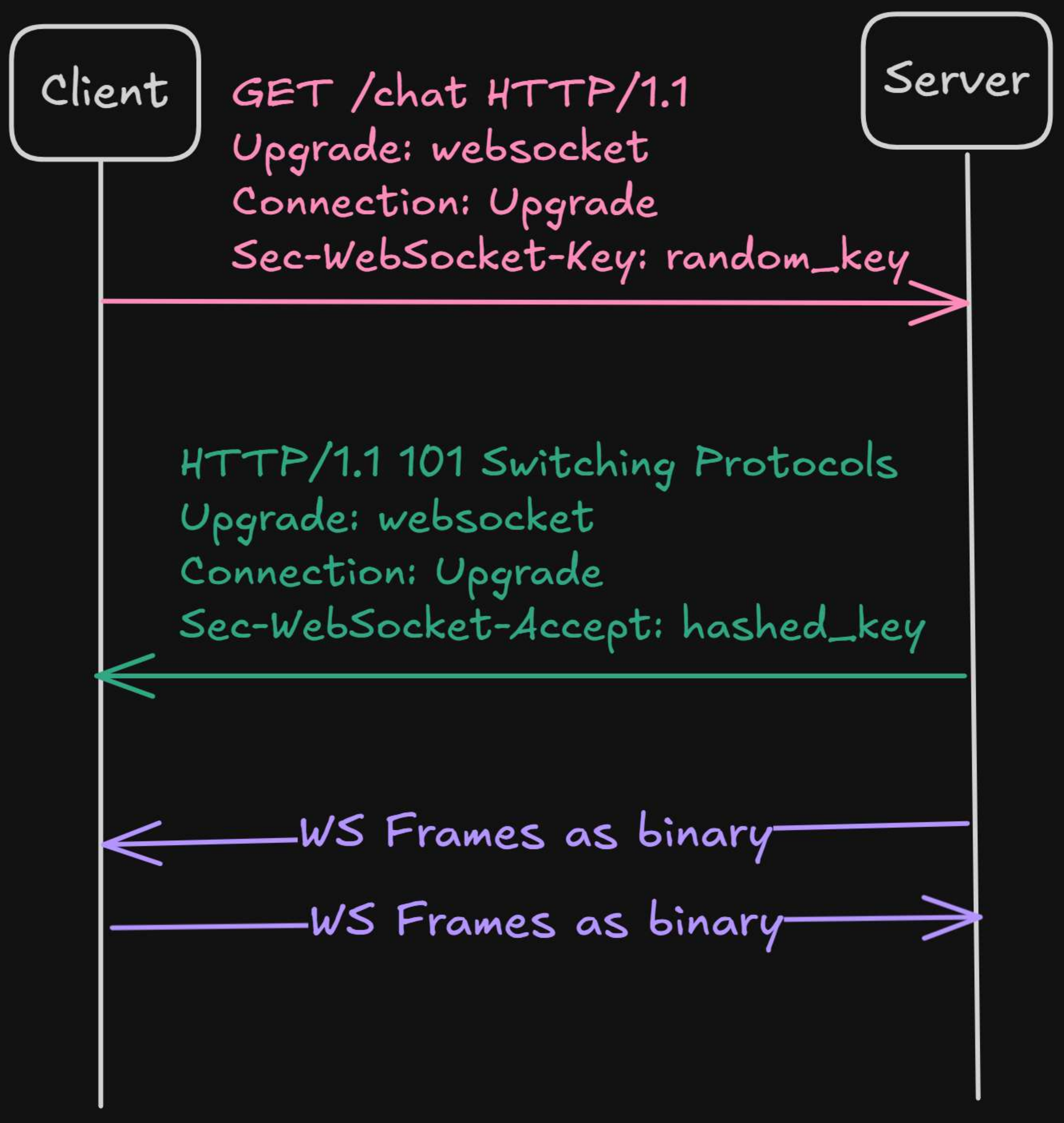
Now both sides can send:

Text frames
Binary frames
Ping / Pong
Close frames

And:

Server can send without waiting
Client can send anytime

-> This is full-duplex



How a WebSocket Connection Closes — Step by Step

WebSocket closes via a small, graceful handshake using CLOSE frames — not by just killing TCP.

Step 0 — Connection is active

TCP connection is open
WebSocket frames are flowing (text/binary)
Ping/Pong may be happening

Step 1 — One side decides to close

Either side can start:

Client (browser refresh, tab close)
Server (shutdown, idle timeout, deploy)

-> There is no "request" or "response" here — only frames.

Step 2 — Initiator sends a CLOSE frame

The initiator sends a WebSocket CLOSE frame.

CLOSE frame contains:

Close code (optional but recommended)
Reason string (optional)

Example:

Opcode: CLOSE
Code: 1000
Reason: Normal closure

At this moment:

Initiator will NOT send any more data frames

Only allowed frames after this:
→ Close / Pong (protocol rule)

Step 3 — Receiver gets CLOSE frame

The other side receives the CLOSE frame and must:

-> Stop sending normal data

-> Send its own CLOSE frame back
(echoing or choosing its own code)

This is mandatory for a graceful shutdown.

Step 4 — TCP connection is closed

After both sides have exchanged CLOSE frames:

-> TCP connection is closed

-> File descriptors released

-> Memory cleaned up

--> WebSocket is fully closed
--> No resource leak

Client —— CLOSE ——> Server
Client <— CLOSE —— Server
TCP connection closed

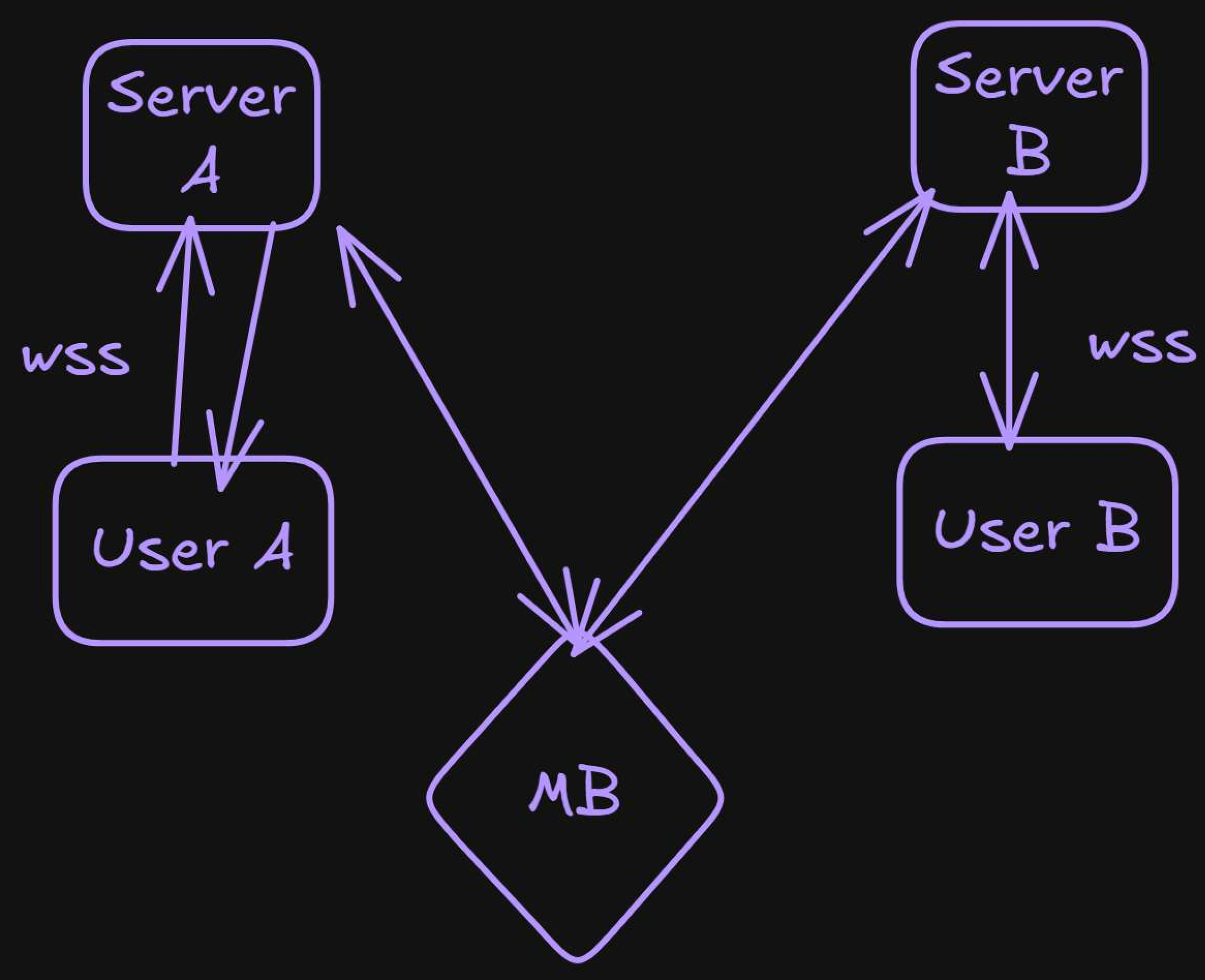
Scaling WebSockets

Problem:

WebSocket is stateful
Connection tied to one server
Load balancer must use sticky sessions

Scaling Solutions:

-> Sticky load balancing
-> Shared pub/sub (Redis, Kafka, etc.)
-> Connection registries
-> Horizontal scaling with event fan-out



Chat App

Failure Handling in WebSockets

Network drops → connection lost

Server crash → clients disconnect

Common strategies:

-> Heartbeats (Ping/Pong)

-> Auto reconnect on client

-> Resume sessions

-> Message acknowledgements (app-level)

--> WebSocket itself does not guarantee delivery

When You SHOULD Use WebSockets

-> Chat systems
-> Live notifications
-> Real-time dashboards
-> Online games
-> Collaborative editors
-> Live trading apps

When You SHOULD NOT Use WebSockets

-> Simple CRUD APIs
-> Rare updates
-> Stateless REST services
-> If SSE is enough (one-way)

DEMO