Real-time Communication Guide for System Design Interviews

WebSockets, Long Polling, SSE - High-Level Design Focus

Table of Contents

Introduction

Real-time communication enables instant, bidirectional data exchange between clients and servers, essential for interactive modern applications.

Why Real-time Communication Matters

Key Benefits:

- **Instant Updates**: No manual refresh, immediate data sync
- **Better User Experience**: Live interaction, instant feedback
- **Reduced Server Load**: Push updates vs constant polling
- **Lower Latency**: Sub-second response times
- **Efficient**: Persistent connections vs repeated handshakes

Common Use Cases in Interviews

```
1. Chat Applications (WhatsApp, Slack)
   - Instant messaging
   - Typing indicators
   - Read receipts
```

```
   - Presence (online/offline)

2. Collaboration (Google Docs, Figma)
   - Real-time editing
   - Live cursors
   - Change synchronization
   - Conflict resolution

3. Live Updates (Stock prices, Sports)
   - Price tickers
   - Score updates
   - Breaking news
   - Dashboard metrics

4. Gaming (Multiplayer)
   - Player actions
   - Game state sync
   - Leaderboards
   - Match coordination

5. Monitoring & IoT
   - Sensor streams
   - Alert notifications
   - System dashboards
   - Device status
```

Communication Patterns Overview

Evolution of Real-time Communication

```
1. Traditional HTTP (1991)
   Request → Response → Done
   - Stateless
   - New connection each time
   - High overhead
   - No real-time capability

2. Short Polling (Early 2000s)
   Repeated requests every N seconds
   - Simple implementation
   - High latency (average N/2)
   - Wasted requests (90%+)
   - Server load

3. Long Polling (Mid 2000s)
   Request held until data available
   - Lower latency (< 1s)
   - No wasted requests
   - Connection timeout handling
```

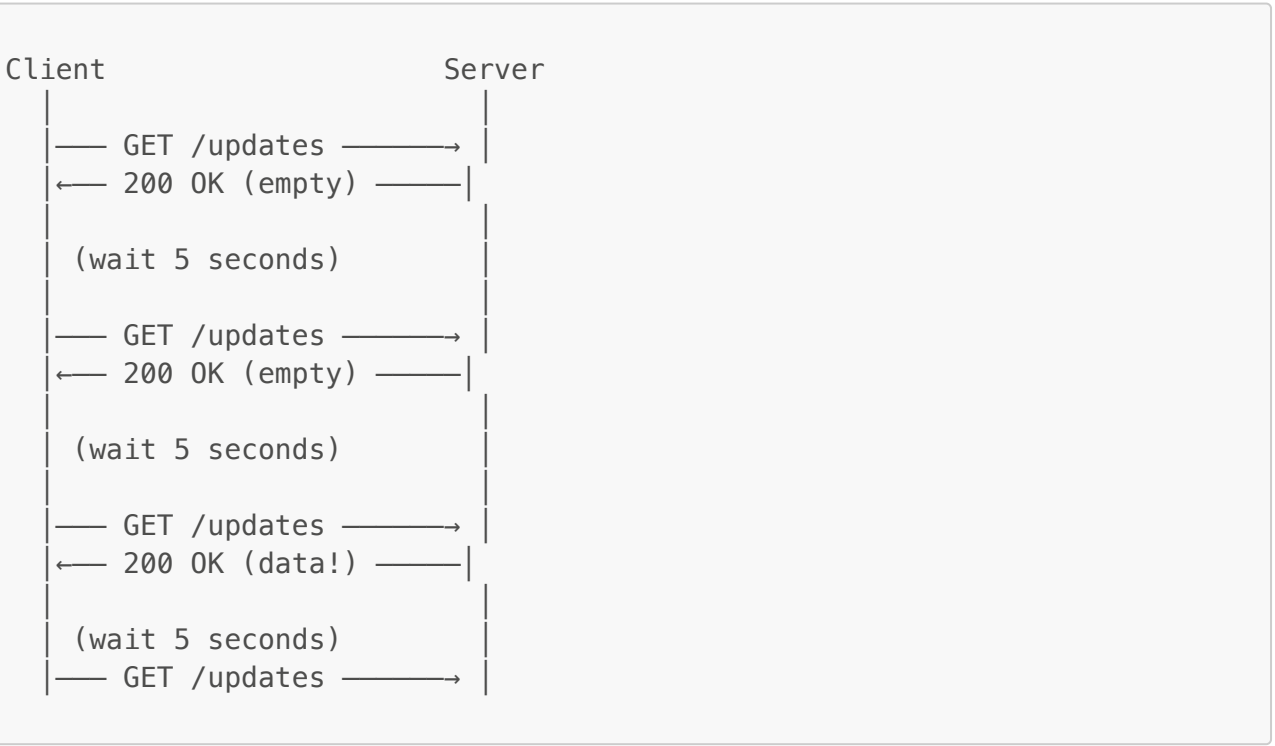
```
   - Still HTTP overhead

4. WebSockets (2011)
   Persistent bidirectional connection
   - Full-duplex
   - Very low latency (< 50ms)
   - Minimal overhead (2-6 bytes)
   - Complex scaling

5. Server-Sent Events (HTML5)
   Server push over HTTP
   - Unidirectional (server→client)
   - Auto-reconnect
   - Simpler than WebSocket
   - Event-based
```

Short Polling

Concept

Client repeatedly sends requests at fixed intervals (e.g., every 5 seconds) to check for updates.

Flow Diagram



Characteristics

```
Latency:
- Average: polling_interval / 2
- Example: 5s interval = 2.5s average latency
```

```
- Max: polling_interval

Efficiency:
- Wasted requests: Very high (90%+)
- Server load: High (constant requests)
- Network usage: High

Resource Usage per 1000 Users (5s polling):
- Requests/sec: 200
- Bandwidth: High (HTTP headers each time)
- Server threads: 200 concurrent
- Database queries: 200/sec
```

When to Use

```
✅ GOOD FOR:
- Low-frequency updates (minutes)
- Simple requirements
- Legacy infrastructure
- Development simplicity
- Not real-time critical

Examples:
- Email inbox (check every minute)
- Weather updates
- Non-critical news feeds

❌ AVOID FOR:
- Real-time chat
- Live collaboration
- High-frequency updates
- Many concurrent users
- Mobile (battery drain)
```

Long Polling

Concept

Client sends request, server holds connection open until data is available or timeout occurs, then client immediately reconnects.

Flow Diagram

```
Client                    Server
  │                         │
  │─── GET /updates ──────→ │
  │                         │ (Server waits...
```

```
    |                          |  monitoring for
    |                          |  new data)
    |  (30-60 seconds)         |
    |                          | Event occurs!
    |←── 200 OK (data!) ──────|
    |                          |
    |─── GET /updates ────────→ | (Immediately)
    |                          | (Server waits...)
    |                          |
```

Characteristics

```
Latency:
- Average: < 1 second
- Near real-time
- Depends on event occurrence

Efficiency:
- Wasted requests: Near 0%
- Server load: Medium (held connections)
- Network usage: Medium

Resource Usage per 1000 Users:
- Held connections: 1000
- Requests/sec: ~Event frequency
- Memory: 1000 * connection_buffer
- Efficient when updates sparse

Connection Management:
- Timeout: 30-60 seconds (reconnect)
- Server must track held connections
- Handle disconnections gracefully
- Scalable to 10K-100K users per server
```

When to Use

```
✅ GOOD FOR:
- Moderate update frequency
- Server→Client updates primarily
- Near real-time needs
- Existing HTTP infrastructure
- Behind proxies/firewalls

Examples:
- Live notifications
- Order status tracking
- Auction bidding
- Chat (acceptable latency)
```

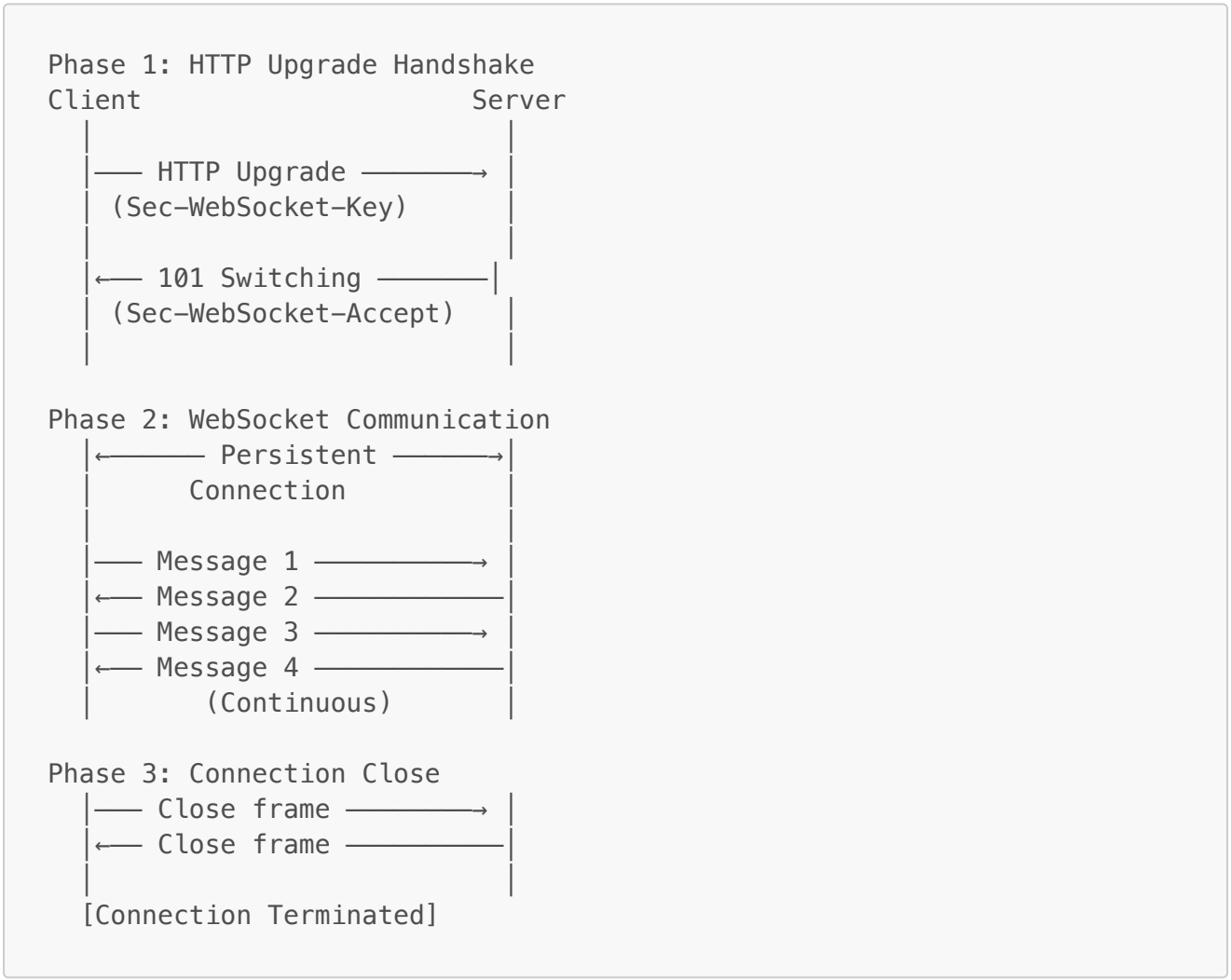
```
- Real-time feeds

❌ AVOID FOR:
- Very high frequency (ms level)
- True bidirectional needed
- Gaming
- Collaborative editing
- Strict latency requirements
```

WebSockets

Concept

Persistent, bidirectional, full-duplex connection established via HTTP upgrade, then uses lightweight framing protocol.

Connection Lifecycle

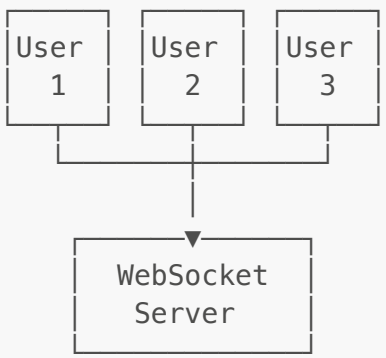


Protocol Advantages

```
Efficiency:
- HTTP Headers: 500-1000 bytes per request
- WebSocket Frame: 2-6 bytes overhead
- Savings: 99% overhead reduction
- Binary data support

Performance:
- Latency: 10-50ms (round-trip)
- Throughput: 1000s messages/sec per connection
- Concurrent connections: 10K-100K per server
- Memory per connection: 5-10KB

Message Types:
- Text frames (UTF-8, typically JSON)
- Binary frames (efficient for media)
- Control frames (ping/pong, close)
```

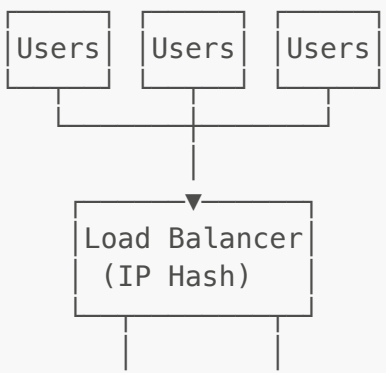
Architecture Patterns

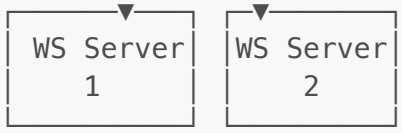
Pattern 1: Simple Single Server




```
Capacity: ~10K connections
Use: Small apps, prototypes
```

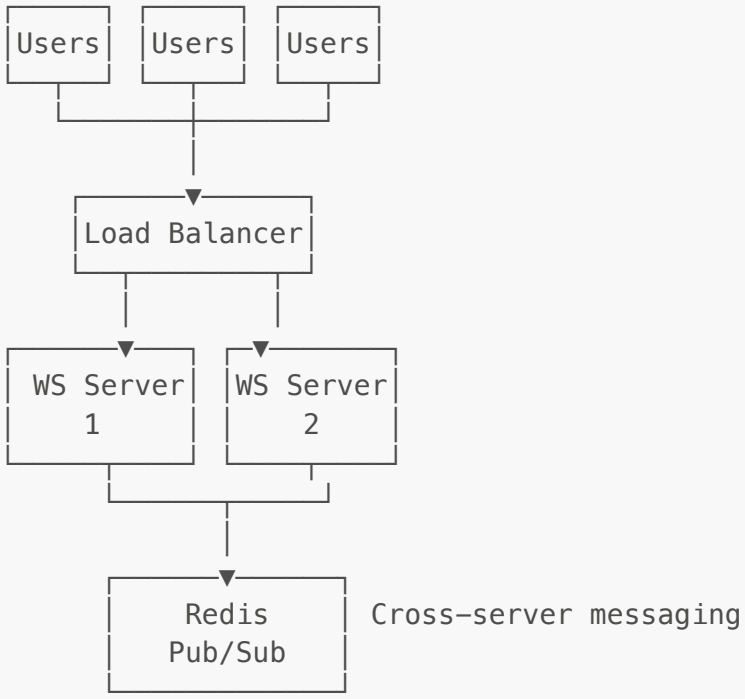
Pattern 2: Load Balanced with Sticky Sessions






```
Challenge: Users on Server 1 can't message users on Server 2
Solution: Need message broker (next pattern)
```

Pattern 3: With Message Broker (Best Practice)




```
Cross-Server Communication:
1. User on Server 1 sends message
2. Server 1 publishes to Redis
3. Redis broadcasts to all servers
4. Both servers push to their clients
5. All users receive message
```

When to Use

```
✅ USE WEBSOCKETS FOR:
- Chat/messaging (WhatsApp, Slack)
- Multiplayer games
- Collaborative editing (Google Docs)
- Live trading platforms
- Real-time dashboards (high frequency)
- Interactive applications
- Video call signaling
```

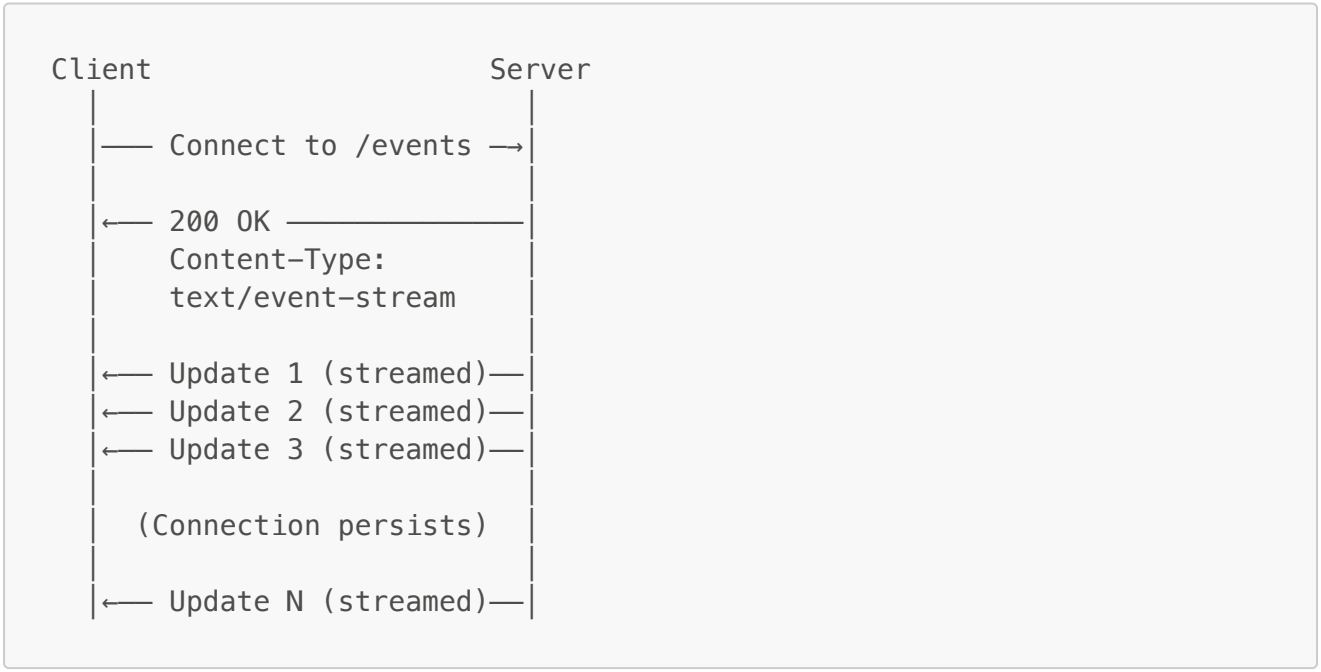
```
Requirements Indicating WebSocket:
- Bidirectional communication
- Low latency (< 100ms)
- High frequency updates
- Real-time interaction
- Both client and server send messages

❌ AVOID WEBSOCKETS FOR:
- Unidirectional updates only
- Low update frequency
- Simple use cases
- Limited scaling budget
- Strong caching needs
```

Server-Sent Events (SSE)

Concept

HTTP connection that remains open, allowing server to push multiple updates to client over time. Unidirectional (server to client only).

Flow Diagram



Key Features

```
1. Automatic Reconnection
   - Browser automatically reconnects if dropped
   - Server can specify retry interval
   - Event IDs for resuming from last received

2. Event Types
```

```
   - Named events for different message types
   - Client registers handlers per type
   - Organized message handling

3. Text-Based Protocol
   - UTF-8 text only
   - Typically JSON
   - Human-readable

4. HTTP-Based
   - Works with existing infrastructure
   - Firewall friendly
   - Proxy compatible
   - Standard HTTP features
```

Architecture

```
Simple SSE Architecture:

┌─────────────┐
│ Clients     │
│ (1000s)     │
└──────┬──────┘
       │
┌──────▼──────┐
│    SSE      │
│ Servers     │
└──────┬──────┘
       │
┌──────▼──────┐
│  Data       │  Event source
│ Source      │
└─────────────┘

Scaled SSE with Pub/Sub:

┌─────────────┐
│ Clients     │
└──────┬──────┘
       │
┌──────▼──────┐
│    CDN      │  Cache layer (optional)
└──────┬──────┘
       │
┌──────▼──────┐
│    SSE      │
│ Servers     │
└──────┬──────┘
       │
┌──────▼──────┐
```

When to Use

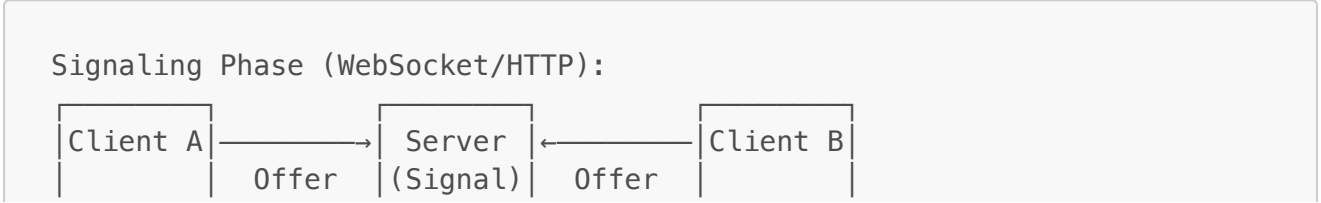
```
✅ USE SSE FOR:
- Stock/crypto price updates
- Live sports scores
- News/social media feeds
- Progress bars for long operations
- Server monitoring dashboards
- Notification streams
- Activity feeds

Requirements Indicating SSE:
- Server→Client updates only
- Text-based data
- Moderate frequency (seconds)
- Automatic reconnection important
- Simple implementation preferred

❌ AVOID SSE FOR:
- Client needs to send frequent updates
- Binary data required
- True bidirectional needed
- IE/old Edge support required
- 6 concurrent connection limit is issue
```

WebRTC

Concept

Peer-to-peer communication protocol for real-time audio, video, and data transfer between browsers without server intermediary.

Architecture



```
┌─────────┐←──────────┐         ┌──────────→┌─────────┐
│         │  Answer   │         │  Answer   │         │
└─────────┘           └─────────┘           └─────────┘

P2P Data Transfer:
┌─────────┐                                 ┌─────────┐
│Client A │←─── Direct P2P Data ──────────→│Client B │
│         │     (Audio/Video/Data)          │         │
└─────────┘                                 └─────────┘

Server only for:
- Initial connection (signaling)
- NAT traversal (STUN/TURN)
- Not involved in media transfer!
```

Components

```
1. Signaling Server
   - Exchange connection information
   - WebSocket or HTTP
   - SDP (Session Description Protocol)
   - ICE candidates

2. STUN Server (Session Traversal Utilities for NAT)
   - Discover public IP/port
   - NAT type detection
   - Free/public available
   - Google: stun.l.google.com:19302

3. TURN Server (Traversal Using Relays around NAT)
   - Relay traffic if P2P fails
   - Fallback for restrictive networks
   - Bandwidth intensive
   - Cost consideration

4. Data Channel
   - P2P data transfer
   - Arbitrary data
   - Low latency
   - Reliable or unreliable mode
```

When to Use

```
✅ USE WEBRTC FOR:
- Video conferencing (Zoom, Google Meet)
- Voice calls (WhatsApp, Discord)
- Screen sharing
- File transfer (P2P)
```

```
- Live streaming (broadcaster→viewers)
- Gaming (P2P actions)

Benefits:
✅ No server bandwidth for media
✅ Very low latency
✅ Encrypted by default
✅ Adaptive quality
✅ Browser native

❌ AVOID WEBRTC FOR:
- Traditional text chat (use WebSocket)
- Server-mediated data
- Simple notifications
- No P2P possible (corporate networks)
- Large group calls (> 10 participants)
```

Comparison Matrix

Detailed Comparison

Feature	Short Poll	Long Poll	WebSocket	SSE	WebRTC
Connection	New each	Held open	Persistent	Held open	P2P
Direction	Request Response	Request Response	Both directions	Server→Client	Both
Latency	2-10s	<1s	<50ms	<200ms	<30ms
Overhead/Msg	500B+	500B+	2-6B	20B+	Minimal
Scalability	Easy	Medium	Hard	Medium	N/A
Server Load	High	Medium	Medium	Medium	Low
Implementation	Simple	Medium	Complex	Simple	Complex
Browser Support	All	All	Modern	Modern	Modern
Firewall Issues	No	No	Some	No	Yes
Binary Data	No	No	Yes	No	Yes
Auto-Reconnect	Yes	No	No	Yes	No

Use Case Matrix

Requirement	Best Choice
Bidirectional	WebSocket
Unidirectional	SSE or Long Polling
Low latency (<50ms)	WebSocket or WebRTC
Medium latency (<1s)	Long Polling or SSE
High latency OK	Short Polling
Binary data	WebSocket or WebRTC
Text only	SSE or Long Polling
Simple infra	Long Polling
Scale to millions	Long Polling or SSE
Media streaming	WebRTC

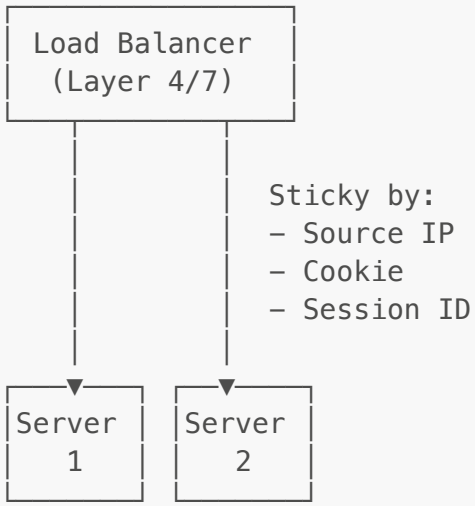
Scaling WebSocket Systems

Challenge: Stateful Connections

```
Problem:
- WebSocket = persistent connection
- Connection tied to specific server
- Can't freely move connections
- Load balancing complex

Traditional Stateless (Easy):
Request 1 → Server A
Request 2 → Server B (different server, no problem)
Request 3 → Server A

WebSocket Stateful (Hard):
Initial → Server A → Connection established
All future → MUST go to Server A
If Server A fails → Connection lost
```

Scaling Solutions

Solution 1: Sticky Sessions

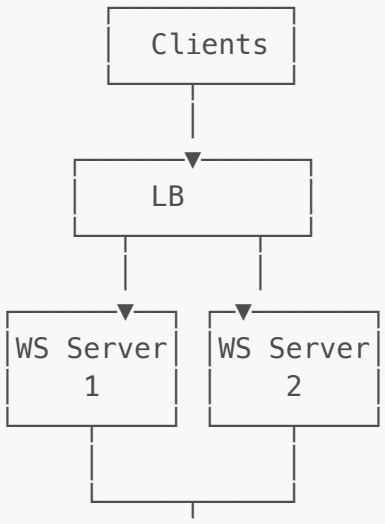


Pros:
✅ Simple concept
✅ Works with any WebSocket server

Cons:
❌ Uneven distribution
❌ Connection draining complex
❌ Failover loses connections
❌ No cross-server messaging

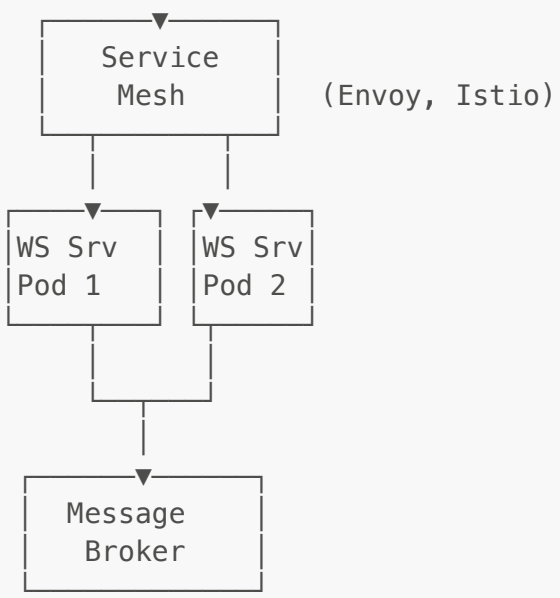
Capacity:
- 10 servers * 10K connections = 100K total

Solution 2: Message Broker Pattern




```
Message Flow:
1. Client A (Server 1) sends message
2. Server 1 → Redis Pub/Sub (real-time)
3. Server 1 → Kafka (persistence)
4. Redis → All Servers (fanout)
5. All Servers → Their clients
6. Kafka Consumer → Database (async)

Pros:
✅ Cross-server communication
✅ True horizontal scaling
✅ Message persistence
✅ No sticky sessions needed

Cons:
❌ Added latency (10-30ms for Redis hop)
❌ More infrastructure
❌ Operational complexity

Scaling Numbers:
- 100 WS servers
- 10K connections per server
- 1M total concurrent users
- Redis: <10ms added latency
- Kafka: Async (no latency impact)
```

Solution 3: Service Mesh Pattern




```
Features:
- Automatic service discovery
- Load balancing
- Health checks
- Circuit breaking
- Observability

Best for:
- Kubernetes environments
- Microservices
- Cloud-native apps
```

Connection State Management

```
Challenge: Where to store connection state?

Option 1: In-Memory (Each Server)
┌──────────────────────┐
│ WS Server 1          │
│ connections = {      │
│   'user-1': socket,  │
│   'user-2': socket   │
│ }                    │
└──────────────────────┘

Pros: Fast
Cons: Lost on restart, not shared

Option 2: Redis (Shared State)
┌──────────────────────────┐
│ Redis                    │
│ user:123 → {server: 'ws-1', │
│             rooms: [...]}   │
```

```
└──────────────────────────────────┘

Pros: Shared, persistent
Cons: Network latency

Option 3: Hybrid
- Active connections: In-memory
- Metadata: Redis
- History: Database

Best Practice for Scale
```

Common Interview Scenarios

1. Chat Application (WhatsApp/Slack-like)

```
REQUIREMENTS:
- 10M daily active users
- 1M concurrent connections
- Real-time messaging (< 100ms)
- Group chats (up to 256 members)
- Message history
- Typing indicators
- Read receipts
- Media sharing

HIGH-LEVEL ARCHITECTURE:

┌─────────────────────────────────────────┐
│           Client Layer                  │
│   Web Browsers, Mobile Apps             │
└─────────────────┬───────────────────────┘
                  │
┌─────────────────▼───────────────────────┐
│         Load Balancer Layer             │
│   DNS → Global LB → Regional LBs        │
└─────────────────┬───────────────────────┘
                  │
┌─────────────────▼───────────────────────┐
│       WebSocket Server Layer            │
│   100 servers, 10K conn each            │
└──┬──────────────────────────────────┬───┘
   │                                  │
┌──▼──────────┐                  ┌────▼──────┐
│    Redis    │                  │   Kafka   │
│   Pub/Sub   │                  │  (Events) │
│  (Real-time)│                  │(Persistent)│
└─────────────┘                  └─────┬─────┘
                                       │
```

KEY DESIGN DECISIONS:

1. WebSocket Choice:
   Why: Bidirectional, low latency, real-time
   Alternative: Long polling (higher latency)
   Trade-off: Complexity vs performance

2. Message Flow:
   Send Path:
   Client → WS Server → Redis (fanout) + Kafka (persist)

   Receive Path:
   Redis → WS Servers → Clients
   Kafka → Cassandra (async)

3. Partitioning Strategy:
   - Partition by userId for 1-1 chats
   - Partition by roomId for groups
   - Ensures message ordering

4. Scaling Numbers:
   - 1M concurrent / 100 servers = 10K per server
   - Redis handles 100K msg/sec fanout
   - Kafka handles 50K msg/sec persistence
   - Cassandra handles 30K writes/sec

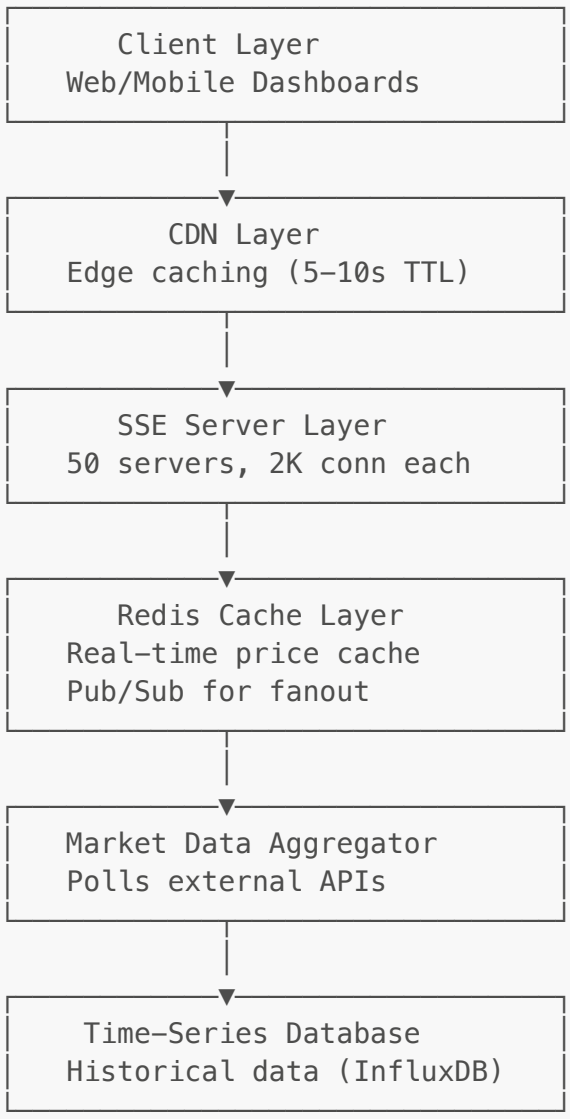
5. Failure Handling:
   - WS server fails → Client auto-reconnects
   - Redis fails → Fallback to Kafka
   - Kafka fails → Redis continues (eventual consistency)
   - Messages queued until systems recover

6. Monitoring:
   - Connection count per server
   - Message delivery latency (P50, P95, P99)
   - Redis pub/sub lag
   - Kafka consumer lag
   - Error rates

INTERVIEW TALKING POINTS:

"I'll use WebSockets for real-time bidirectional chat.
For 1M concurrent users, I'll deploy 100 servers with
10K connections each. Redis Pub/Sub handles real-time
message fanout across servers (< 20ms latency). Kafka
persists all messages for history/audit. Cassandra
stores searchable message history. Partition by userId
for 1-1 chats, roomId for groups to ensure ordering."

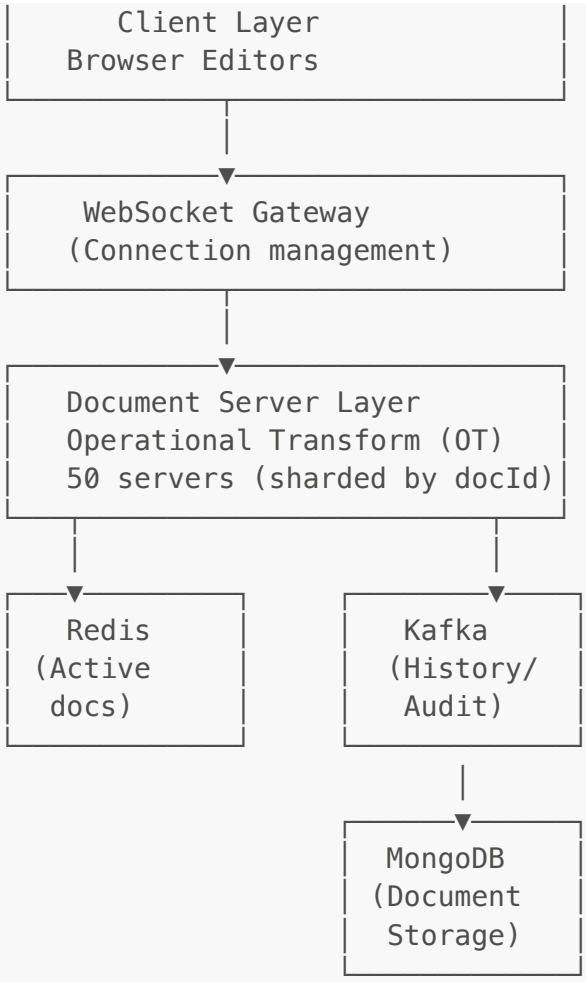
2. Live Stock Price Dashboard

```
REQUIREMENTS:
- 100K concurrent viewers
- 1000+ stocks tracked
- Price updates every second
- Historical charts
- Low latency
- Cost-effective

HIGH-LEVEL ARCHITECTURE:




KEY DESIGN DECISIONS:

1. SSE vs WebSocket:
   Why SSE:
   - Unidirectional (server→client only)
   - Simpler implementation
```

```
   - Automatic reconnection
   - Lower server resources
   - Better for broadcasting

2. Update Strategy:
   - Subscribe only to watched symbols
   - Server throttles to max 1 update/sec
   - Batch multiple symbol updates
   - Client-side interpolation

3. Caching Strategy:
   - Redis: Sub-second cache (<100ms)
   - CDN: 5-10 second cache
   - Reduces SSE server load 80%

4. Scaling Numbers:
   - 100K users / 50 servers = 2K per server
   - 1K symbols * 1 update/sec = 1K updates/sec
   - Redis handles 100K ops/sec easily
   - CDN offloads 80% of requests

5. Cost Optimization:
   - CDN caching saves 80% bandwidth
   - Throttle per-user subscriptions (max 50 symbols)
   - Aggregate updates in batches
   - Use compression

INTERVIEW TALKING POINTS:

"For stock prices, I'll use SSE since updates are
unidirectional. 50 SSE servers handle 2K connections
each. Market data aggregator updates Redis every second.
Redis pub/sub fans out to SSE servers. CDN caches with
5s TTL reduces load by 80%. Clients subscribe only to
symbols they're watching. This is simpler and cheaper
than WebSocket since we don't need bidirectional."
```

3. Collaborative Document Editor (Google Docs-like)

```
REQUIREMENTS:
- Multiple users editing simultaneously
- Real-time cursor positions
- Character-by-character updates
- Conflict resolution
- 50K active documents
- 10 concurrent editors per document

HIGH-LEVEL ARCHITECTURE:

┌──────────────────────────────────────┐
```



KEY DESIGN DECISIONS:

1. Why WebSocket:
   - Bidirectional: Users send edits, receive updates
   - Low latency: < 100ms for edit to visible
   - High frequency: Character-by-character

2. Document Sharding:
   - Documents assigned to servers by docId
   - Hash-based: hash(docId) % serverCount
   - All editors of same document → same server
   - Simplifies conflict resolution

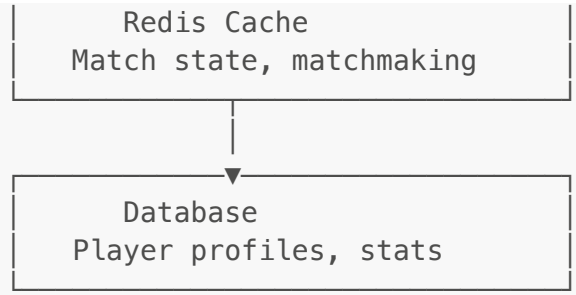
3. Operational Transform:
   - Handle concurrent edits
   - Transform operations to maintain consistency
   - Server is authoritative
   - Example: Two users insert at same position
     → Operations transformed to maintain intent

4. State Management:
   - Active documents: Redis (fast access)
   - Document operations: Kafka (ordered log)
   - Final documents: MongoDB (persistence)
   - LRU cache for hot documents

```
5. Cursor Broadcasting:
   - Throttled to 10 updates/sec per user
   - Broadcast to room, not persisted
   - Includes: position, color, userId

6. Scaling Numbers:
   - 50K documents / 50 servers = 1K per server
   - 10 editors * 50 ops/sec = 500 ops/sec per doc
   - Redis caches 10K hot documents
   - MongoDB stores all documents

INTERVIEW TALKING POINTS:

"WebSocket for real-time bidirectional editing. Shard
documents by docId so all editors connect to same server -
simplifies OT conflict resolution. Redis caches active
document state. Kafka logs all operations for replay/audit.
MongoDB persists final documents. Throttle cursor updates
to 10/sec to reduce bandwidth. Server applies Operational
Transform to resolve concurrent edits."
```

4. Multiplayer Game

```
REQUIREMENTS:
- 100K concurrent players
- 10K active matches (10 players each)
- Ultra-low latency (< 50ms)
- Game state synchronization
- Cheat prevention
- 60 FPS (16ms per frame)

HIGH-LEVEL ARCHITECTURE:

┌──────────────────────────────────┐
│      Client Layer                │
│   Game Clients (Unity/Unreal)    │
└──────────────┬───────────────────┘
               │
┌──────────────▼───────────────────┐
│    WebSocket Gateway             │
│   (Connection routing)           │
└──────────────┬───────────────────┘
               │
┌──────────────▼───────────────────┐
│    Game Server Layer             │
│   Authoritative servers          │
│   1K servers, 10 games each      │
└──────────────┬───────────────────┘
               │
┌──────────────▼───────────────────┐
```



KEY DESIGN DECISIONS:

1. Why WebSocket:
   - Lowest latency option
   - Bidirectional (client actions, server updates)
   - Binary protocol support (efficient)
   - 60 updates/sec requires persistent connection

2. Server-Authoritative Model:
   - Client sends inputs only
   - Server simulates game
   - Server broadcasts state
   - Prevents cheating

3. Game Loop (Server-side):
   - 60 ticks per second (16ms per tick)
   - Process inputs from all players
   - Update physics/collision
   - Broadcast state to clients
   - State includes: positions, health, actions

4. Client-Side Prediction:
   - Client predicts movement locally
   - Immediate visual feedback
   - Server state is authoritative
   - Client reconciles differences

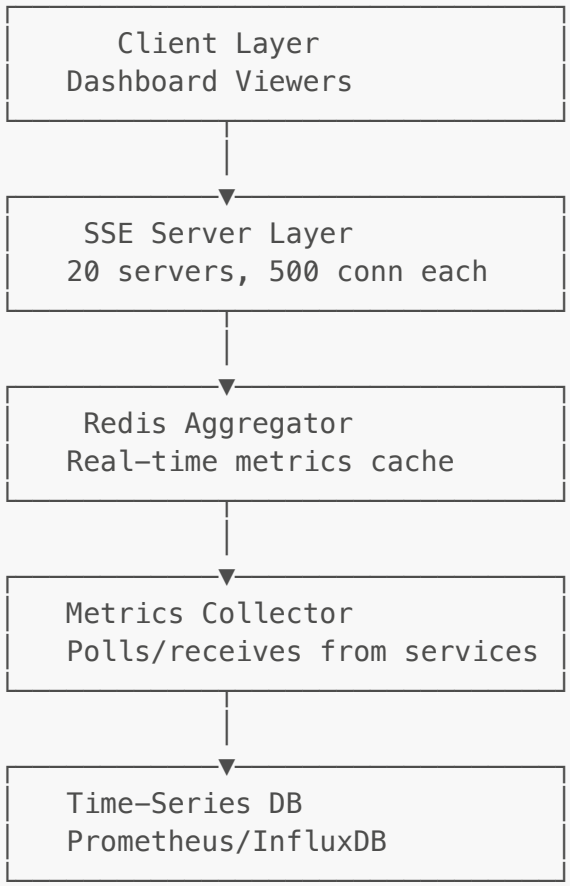
5. Optimization Techniques:
   - Binary protocol (smaller messages)
   - Delta compression (send only changes)
   - Interest management (nearby players only)
   - State interpolation (smooth movement)
   - Dead reckoning (predict between updates)

6. Scaling Numbers:
   - 10K matches / 1K servers = 10 matches per server
   - 10 players * 10 matches = 100 connections per server
   - 60 updates/sec * 100 players = 6K msgs/sec per server
   - Binary messages: ~100 bytes each
   - Bandwidth: 600KB/sec per server

7. Matchmaking:
   - Redis stores waiting players
   - Match players by skill/region

```
   - Assign to least-loaded server
   - Create game instance

INTERVIEW TALKING POINTS:

"WebSocket for ultra-low latency gaming. Server-authoritative
model prevents cheating. Game servers run at 60 FPS,
broadcasting state to clients. Clients predict movement
locally for responsive feel, then reconcile with server.
Binary protocol reduces message size. Delta compression
sends only changes. Interest management limits updates to
nearby players. 1K game servers handle 10 matches each."
```

5. Live Dashboard (System Monitoring)

```
REQUIREMENTS:
- 10K concurrent viewers
- Metrics from 1000 services
- Update every 5 seconds
- Historical charts
- Alerting
- Multiple dashboards per user

HIGH-LEVEL ARCHITECTURE:



```

```
KEY DESIGN DECISIONS:

1. SSE vs WebSocket:
   Why SSE:
   - Unidirectional (server→client)
   - Updates every 5 seconds (not ultra-low latency)
   - Automatic reconnection
   - Simpler than WebSocket
   - Lower resource usage

2. Subscription Model:
   - Clients subscribe to specific dashboards
   - Server filters relevant metrics
   - Reduces unnecessary data transfer

3. Aggregation Strategy:
   - Metrics collector aggregates from 1000 services
   - Updates Redis every 5 seconds
   - Redis pub/sub notifies SSE servers
   - SSE servers stream to subscribed clients

4. Scaling Numbers:
   - 10K viewers / 20 servers = 500 per server
   - 1000 services * 100 metrics = 100K metrics
   - Update interval: 5 seconds
   - Data volume: 20K updates/sec

5. Optimization:
   - Subscribe only to visible metrics
   - Downsample historical data
   - Compress metric payloads
   - CDN for static dashboard UI

INTERVIEW TALKING POINTS:

"SSE for unidirectional metrics streaming. Simpler than
WebSocket since clients don't send data. Metrics collector
aggregates from services, updates Redis every 5s. Redis
pub/sub fans out to 20 SSE servers. Clients subscribe to
specific dashboards. Automatic reconnection handles network
issues. Time-series DB stores history for charts."
```

Design Patterns

1. Heartbeat/Keepalive Pattern

```
Purpose: Detect dead connections

┌──────────┐                    ┌──────────┐
```

```
┌──────────┐                    ┌──────────┐
│ Client   │                    │ Server   │
└──────────┘                    └──────────┘
     │                               │
     │                               │
     │←──── Ping (every 30s) ────────│
     │──────── Pong ─────────────────→│
     │                               │
     │  (If no pong received,        │
     │   connection is dead)         │
```

```
Implementation Strategy:
- Server sends ping every 30 seconds
- Client must respond with pong
- If no pong in 60 seconds → terminate
- Prevents resource leak from dead connections

Benefits:
✅ Clean up zombie connections
✅ Free up server resources
✅ Accurate connection counts
✅ Better monitoring
```

2. Reconnection with Exponential Backoff

```
Pattern: Retry with increasing delays

Connection Attempts:
Attempt 1: Wait 1 second
Attempt 2: Wait 2 seconds
Attempt 3: Wait 4 seconds
Attempt 4: Wait 8 seconds
Attempt 5: Wait 16 seconds
...
Max: Wait 30 seconds

With Jitter (Randomization):
- Prevents thundering herd
- Adds randomness to delay
- Spreads reconnection load

Example:
Base delay: 4 seconds
With jitter: 3–5 seconds (random)

Benefits:
✅ Prevents server overload
✅ Handles temporary failures
✅ Gradual recovery
✅ Reduces thundering herd
```

3. Presence Detection Pattern

```
Architecture:

┌──────────┐
│ Client   │ ──── connect ────→ Online
│          │
│          │ ──── activity ───→ Active
│          │
│          │ (30s no activity)→ Away
│          │
│          │ ──── disconnect ─→ Offline
└──────────┘

State Management:
┌──────────────────────────────┐
│ Redis                        │
│ user:123:status = "online"   │
│ user:123:lastSeen = <time>   │
│ user:123:server = "ws-5"     │
└──────────────────────────────┘

Presence Broadcast:
- User connects → Broadcast "online"
- User disconnects → Broadcast "offline"
- Idle timeout → Broadcast "away"
- Periodic heartbeat updates lastSeen

Scaling:
- Redis stores presence for all users
- TTL on presence keys (5 minutes)
- If no heartbeat → auto-expire → offline
```

4. Message Ordering Pattern

```
Challenge: Ensure message order in distributed system

Solution 1: Sequence Numbers
Message Structure:
{
  userId: "user-123",
  sequence: 42,
  timestamp: 1642098765,
  content: "Hello"
}

Client tracks: lastReceivedSequence = 41
Receives: sequence 42 → Process
Receives: sequence 44 → Buffer (wait for 43)
```

```
Receives: sequence 43 → Process 43, then 44

Solution 2: Vector Clocks
Track causality in distributed system
Each server has clock
Compare clocks to determine order

Solution 3: Partition-Based
- Same conversation → same partition
- Partition ensures order
- Kafka/Redis pub/sub maintains order per partition

Best Practice: Partition by conversation/room
```

5. Graceful Shutdown Pattern

```
Process:
1. Server receives shutdown signal
2. Stop accepting new connections
3. Send close frame to all existing connections
4. Wait for clients to acknowledge
5. Flush pending messages
6. Close connections
7. Shutdown complete

Connection Draining:
┌─────────────┐
│New Conns    │ ─X→ Server (rejected)
└─────────────┘


┌─────────────┐
│Existing     │ ────→ Server (maintained)
│Conns        │       (until drained)
└─────────────┘

Graceful Shutdown Time:
- Notify clients: 5 seconds
- Drain connections: 30 seconds
- Force close remaining: After 30s
- Total: ~35 seconds

Benefits:
✅ No dropped messages
✅ Clients can reconnect
✅ Clean deployment
✅ No service disruption
```

How to Use in Interviews

Interview Framework (35-40 minutes)

Phase 1: Requirements Clarification (5 min)

```
CRITICAL QUESTIONS TO ASK:

Scale:
□ How many concurrent users?
□ Expected message frequency?
□ Peak load scenarios?
□ Geographic distribution?

Communication Pattern:
□ Bidirectional or unidirectional?
□ Client→Server or Server→Client or both?
□ Real-time requirements (latency)?
□ Message ordering important?

Data Characteristics:
□ Message size?
□ Text or binary data?
□ Message persistence needed?
□ Historical data requirements?

Non-Functional:
□ Reliability requirements?
□ Failure tolerance?
□ Compliance/security needs?
□ Budget constraints?

EXAMPLE DIALOGUE:

Interviewer: "Design a real-time chat system"

You: "Let me clarify some requirements:
- How many concurrent users are we expecting?
- Is this 1-1 chat, group chat, or both?
- What's our latency requirement - sub-second acceptable?
- Do we need message history and search?
- Any specific reliability requirements?
- Are read receipts and typing indicators needed?"

This shows: Requirements gathering skills, systematic thinking
```

Phase 2: High-Level Design (10-15 min)

```
STEP-BY-STEP APPROACH:
```

```
1. Choose Communication Technology

   Decision Framework:

   Question: Bidirectional?
   ├─ Yes → WebSocket or Long Polling
   └─ No → SSE or Long Polling

   Question: Latency requirement?
   ├─ < 100ms → WebSocket
   ├─ < 1s → Long Polling or SSE
   └─ > 1s → Short Polling

   Question: Update frequency?
   ├─ High (multiple/sec) → WebSocket or SSE
   ├─ Medium (every few sec) → Long Polling or SSE
   └─ Low (minutes) → Short Polling

2. Draw Architecture Boxes

   [Clients] → [LB] → [WS Servers] → [Message Broker] → [Database]

   Add:
   - Connection management layer
   - Message routing layer
   - Persistence layer
   - Caching layer

3. Explain Data Flow

   Write Path:
   Client → WebSocket → Server → Broker → Other Servers → Clients

   Read Path:
   Database → Cache → Server → WebSocket → Client

4. Call Out Key Components

   - Load balancer (sticky sessions vs not)
   - Message broker (Redis/Kafka)
   - Database (type and why)
   - Cache (Redis for state)

EXAMPLE EXPLANATION:

"For a chat application, I'll use WebSocket because we need
bidirectional communication with low latency. Architecture:
Clients connect via load balancer to WebSocket servers.
Messages flow through Redis Pub/Sub for real-time fanout
across servers. Kafka persists messages for history. This
allows horizontal scaling while maintaining cross-server
communication."
```

Phase 3: Deep Dive (15-20 min)

```
TOPICS TO COVER IN DEPTH:

A. Scaling Strategy
   "To scale to 1M concurrent:
   - 100 WebSocket servers
   - 10K connections per server
   - Sticky sessions with IP hash
   - Redis Pub/Sub for cross-server
   - Automatic failover and reconnection"

B. Message Flow Details
   "When user sends message:
   1. Client → WS Server via persistent connection
   2. Server validates and enriches message
   3. Publish to Redis channel for real-time
   4. Publish to Kafka for persistence
   5. Redis fans out to all WS servers
   6. Servers push to connected clients
   7. Kafka consumer writes to database async"

C. Failure Handling
   "Failure scenarios:
   - WS server crash: Clients reconnect to healthy server
   - Redis failure: Use Kafka as fallback (higher latency)
   - Kafka failure: Redis continues (eventual consistency)
   - Message persistence delayed but delivery continues
   - All messages have unique IDs for deduplication"

D. Monitoring & Metrics
   "Key metrics to track:
   - Active connections per server
   - Message latency (P50, P95, P99)
   - Connection success/failure rate
   - Reconnection frequency
   - Message delivery rate
   - Redis pub/sub lag
   - Alerts on: connection spikes, latency > 500ms"

E. Optimization
   "Performance optimizations:
   - Binary protocol for game data
   - JSON for chat (human-readable)
   - Message batching when possible
   - Compression for large payloads
   - Throttle non-critical updates (typing indicators)
   - Regional deployment for lower latency"
```

Phase 4: Trade-offs Discussion (5 min)

```
DISCUSS ALTERNATIVES:

1. WebSocket vs Long Polling
   "I chose WebSocket because:
   ✅ Lower latency (10-50ms vs 500-1000ms)
   ✅ True bidirectional
   ✅ Lower overhead (2-6 bytes vs 500+ bytes)
   ✅ Better for high frequency

   BUT Long Polling could work if:
   - Latency requirement relaxed
   - Simpler ops preferred
   - Scaling budget limited
   - Existing HTTP infrastructure"

2. Redis vs Kafka for Real-time
   "Using both:
   - Redis: Real-time fanout (<10ms latency)
   - Kafka: Reliable persistence, replay

   Could use only Kafka:
   ✅ Simpler infrastructure
   ❌ Higher latency (50-100ms)
   ❌ More complex consumer management"

3. Sticky Sessions vs Service Mesh
   "Started with sticky sessions:
   ✅ Simpler initial implementation
   ✅ Works with standard load balancers
   ❌ Uneven distribution
   ❌ Connection draining complex

   At scale, consider service mesh:
   ✅ Better distribution
   ✅ Auto-scaling
   ✅ Health management
   ❌ More complexity"
```

Trade-offs & Decisions

1. Technology Selection Trade-offs

```
WebSocket Pros & Cons:

Pros:
✅ Lowest latency (10-50ms)
```

```
✅ Bidirectional
✅ Minimal per-message overhead
✅ High frequency capable
✅ Binary data support

Cons:
❌ Complex scaling (stateful)
❌ Requires sticky sessions or message broker
❌ More server resources
❌ Operational complexity
❌ Potential firewall issues

Best for: Chat, gaming, collaboration
Avoid for: Simple notifications, one-way updates

Long Polling Pros & Cons:

Pros:
✅ Works with existing HTTP infra
✅ No special server requirements
✅ Firewall/proxy friendly
✅ Easier to scale than WebSocket
✅ Near real-time (<1s)

Cons:
❌ Higher latency than WebSocket
❌ HTTP overhead per update
❌ Timeout handling complexity
❌ Not truly bidirectional
❌ Reconnection overhead

Best for: Notifications, status updates
Avoid for: High-frequency, gaming

SSE Pros & Cons:

Pros:
✅ Simpler than WebSocket
✅ Automatic reconnection
✅ Event ID for resume
✅ HTTP-based
✅ Good for broadcasting

Cons:
❌ Unidirectional only
❌ Text-based only
❌ Browser connection limits (6)
❌ No IE/old Edge support
❌ HTTP overhead (vs WebSocket)

Best for: Live feeds, dashboards
Avoid for: Bidirectional, binary data
```

2. Scaling Trade-offs

```
Vertical Scaling (Bigger Servers):

Single powerful server:
- Connections: 100K
- CPU: 64 cores
- Memory: 256GB
- Network: 10Gbps

Pros:
✅ Simpler architecture
✅ No cross-server communication
✅ Easier state management

Cons:
❌ Single point of failure
❌ Limited by hardware
❌ Expensive
❌ Deployment downtime

Horizontal Scaling (More Servers):

100 smaller servers:
- Connections: 1K each = 100K total
- CPU: 4 cores each
- Memory: 16GB each
- Network: 1Gbps each

Pros:
✅ Better fault tolerance
✅ Incremental scaling
✅ Cost-effective
✅ Rolling deployments

Cons:
❌ Cross-server communication needed
❌ Distributed state
❌ More complex
❌ Load balancing

Best Practice: Horizontal for production systems
```

3. Persistence Trade-offs

```
Option 1: Memory Only (Redis)

Pros:
```

```
✅ Fastest (<1ms)
✅ Simple
✅ High throughput

Cons:
❌ Data loss on crash
❌ Limited by RAM
❌ No historical queries

Use Case: Temporary data (presence, typing indicators)

Option 2: Disk + Memory (Kafka)

Pros:
✅ Durable
✅ High throughput
✅ Replay capability
✅ Long retention

Cons:
❌ Slightly higher latency (5-10ms)
❌ Storage costs
❌ Operational overhead

Use Case: Message history, audit trail

Option 3: Database (PostgreSQL/Cassandra)

Pros:
✅ Queryable
✅ Structured data
✅ ACID properties
✅ Long-term storage

Cons:
❌ Slower (10-100ms)
❌ Limited throughput
❌ Scaling complexity

Use Case: Historical queries, reports

Hybrid Approach (Best Practice):
- Redis: Active connections, presence
- Kafka: Message stream, replay
- Database: Long-term storage, search
- Each optimized for its purpose
```

Interview Tips

What Interviewers Look For

```
1. Requirements Clarification
   ✅ Asks about scale, latency, reliability
   ✅ Understands bidirectional vs unidirectional
   ✅ Considers trade-offs
   ❌ Jumps to solution immediately

2. Technology Selection
   ✅ Explains WHY choosing specific technology
   ✅ Discusses alternatives
   ✅ Considers constraints
   ❌ Always picks WebSocket without thinking

3. Scalability
   ✅ Plans for horizontal scaling
   ✅ Addresses cross-server communication
   ✅ Discusses connection limits
   ❌ Ignores stateful connection challenges

4. Reliability
   ✅ Handles reconnection
   ✅ Addresses failure scenarios
   ✅ Discusses monitoring
   ❌ Assumes perfect network

5. Depth of Knowledge
   ✅ Understands protocol details
   ✅ Knows latency characteristics
   ✅ Aware of operational challenges
   ❌ Surface-level understanding only
```

Common Interview Questions

```
Q1: "Why use WebSocket over HTTP polling?"

Good Answer:
"WebSocket provides bidirectional, low-latency communication
with minimal overhead (2-6 bytes vs 500+ bytes per message).
For a chat app with 1000 messages per user per day, this
saves significant bandwidth and reduces latency from seconds
to milliseconds. However, WebSocket is more complex to scale
due to stateful connections requiring sticky sessions or
message brokers for cross-server communication."

Q2: "How do you scale WebSocket to millions of users?"

Good Answer:
"Three-layer approach: 1) Horizontal scaling with 100+
servers (10K connections each), 2) Message broker (Redis
Pub/Sub) for cross-server fanout, 3) Load balancer with
```

```
connection awareness. Also need: connection state in Redis,
heartbeat for dead connection detection, auto-scaling based
on connection count, and monitoring for connection lag."

Q3: "WebSocket vs Server-Sent Events?"

Good Answer:
"Depends on requirements. WebSocket if bidirectional
communication needed (chat, gaming). SSE if server→client
only (stock prices, notifications). SSE is simpler with
auto-reconnect and event resumption. WebSocket gives lower
latency and supports binary. For stock dashboard, I'd
choose SSE. For chat, WebSocket."

Q4: "How do you handle connection failures?"

Good Answer:
"Multi-layered approach: Client implements exponential
backoff for reconnection (1s, 2s, 4s delays). Server
includes message IDs so client can request missed messages.
Use Redis to track last received message per user. On
reconnect, client sends lastMessageId, server sends missed
messages from Kafka. Also implement heartbeat to detect
dead connections proactively."
```

Strong Interview Phrases

```
DEMONSTRATE KNOWLEDGE:

✅ "The trade-off between WebSocket and Long Polling is..."
   Shows understanding of alternatives

✅ "At scale, we'd need to consider..."
   Shows scaling awareness

✅ "For monitoring, I'd track metrics like..."
   Shows operational maturity

✅ "The bottleneck would likely be..."
   Shows system thinking

✅ "To handle this failure scenario..."
   Shows reliability focus

✅ "Let me clarify the requirements around..."
   Shows requirements gathering

✅ "We could also use X, but I prefer Y because..."
   Shows decision-making ability
```

```
✅ "This assumes Z. If different, we'd adjust by..."
   Shows flexibility
```

Common Mistakes to Avoid

```
❌ RED FLAGS:

1. "WebSocket is always better"
   Reality: Depends on requirements

2. "Scaling is just adding more servers"
   Reality: Stateful connections are complex

3. "We don't need monitoring"
   Reality: Critical for production

4. "Long Polling is outdated"
   Reality: Still valid for many use cases

5. "I'll use HTTP/2 for real-time"
   Reality: HTTP/2 push not for app data

6. "Sticky sessions solve everything"
   Reality: Creates issues with failover

7. "We can handle infinite connections"
   Reality: Each server has limits

8. "Message order doesn't matter"
   Reality: Often critical (chat, gaming)

9. "We'll figure out reconnection later"
   Reality: Must plan upfront

10. "One technology fits all use cases"
    Reality: Different needs, different solutions
```

Quick Reference Checklist

Design Checklist for Real-time Systems

```
REQUIREMENTS PHASE:
□ Clarify if bidirectional or unidirectional
□ Determine latency requirements (<50ms, <1s, etc.)
□ Understand message frequency
□ Identify scale (concurrent users)
□ Check data type (text vs binary)
```

```
□ Assess persistence needs
□ Consider geographic distribution

TECHNOLOGY SELECTION:
□ Choose between WebSocket, Long Polling, SSE
□ Justify choice with requirements
□ Discuss alternatives considered
□ Consider operational complexity
□ Evaluate cost implications

ARCHITECTURE DESIGN:
□ Define connection management strategy
□ Plan load balancing approach (sticky vs not)
□ Design cross-server communication (if needed)
□ Choose message broker (Redis/Kafka)
□ Plan data persistence layer
□ Design caching strategy

SCALING CONSIDERATIONS:
□ Calculate connections per server
□ Plan horizontal scaling approach
□ Address stateful connection challenges
□ Design for auto-scaling
□ Consider connection limits
□ Plan capacity

RELIABILITY:
□ Design reconnection strategy
□ Implement heartbeat mechanism
□ Plan for graceful degradation
□ Handle network partitions
□ Design for zero-downtime deployments
□ Create failure recovery plan

PERFORMANCE:
□ Minimize message overhead
□ Consider binary vs text protocol
□ Implement message batching (if applicable)
□ Plan for compression
□ Optimize message routing
□ Reduce unnecessary broadcasts

MONITORING:
□ Track active connections
□ Monitor message latency
□ Track connection success/failure rates
□ Monitor broker lag (Redis/Kafka)
□ Set up alerts for anomalies
□ Create dashboards

SECURITY:
□ Use WSS (WebSocket Secure) - TLS/SSL
□ Implement authentication
```

```
□ Validate all messages
□ Rate limit per user
□ Prevent message injection
□ Audit sensitive operations
```

Conclusion

Real-time communication is a critical component in modern system design. Success in interviews requires understanding not just the technologies, but when and why to use them.

Key Takeaways

Technology Selection Framework:

```
Ask yourself:
1. Bidirectional? → WebSocket
2. Unidirectional? → SSE or Long Polling
3. Ultra-low latency? → WebSocket
4. Simple needs? → Long Polling
5. Media streaming? → WebRTC

No single "best" technology - depends on requirements!
```

Scaling Principles:

```
1. WebSocket = Stateful
   - Plan for sticky sessions OR message broker
   - Cannot freely move connections
   - More complex than stateless HTTP

2. Horizontal Scaling Essential
   - Single server limits: 10K-100K connections
   - Use message broker (Redis/Kafka) for cross-server
   - Monitor and auto-scale

3. Connection Management Critical
   - Heartbeat to detect dead connections
   - Graceful shutdown for deployments
   - Reconnection with exponential backoff
```

Interview Success Tips:

```
1. ALWAYS clarify requirements first
   - Latency needs
   - Bidirectional vs unidirectional
```

```
   - Scale expectations
   - Persistence requirements

2. EXPLAIN your technology choice
   - Not just "I'll use WebSocket"
   - But "WebSocket because bidirectional + low latency"
   - Discuss alternatives
   - Mention trade-offs

3. ADDRESS scaling explicitly
   - Don't ignore stateful connection challenge
   - Explain cross-server communication
   - Discuss load balancing
   - Plan for failures

4. SHOW depth with details
   - Latency numbers (10-50ms for WebSocket)
   - Connection limits (10K per server)
   - Overhead comparisons (2-6 bytes vs 500 bytes)
   - Capacity calculations

5. THINK about operations
   - Monitoring strategy
   - Failure scenarios
   - Deployment approach
   - Cost implications
```

Common Interview Scenarios Summary

Scenario	Technology	Key Considerations
Chat App	WebSocket	Redis Pub/Sub, Message ordering
Stock Dashboard	SSE	Throttling, CDN caching
Collaborative Edit	WebSocket	OT/CRDT, Conflict resolution
Multiplayer Game	WebSocket	Server-auth, Binary protocol
Notifications	Long Polling or SSE	Timeout handling, Auto-reconnect
Video Conference	WebRTC	STUN/TURN, P2P bandwidth

Final Interview Framework

```
STEP 1: Requirements (5 minutes)
"Let me clarify: Is this bidirectional? What latency do we need?
How many concurrent users?"

STEP 2: Technology Choice (2 minutes)
"I'll use WebSocket because [reasons]. Alternatively, we could
use [option] but [trade-off]."

STEP 3: Architecture (10 minutes)
"High-level: Clients → Load Balancer → WS Servers → Message
Broker → Database. Let me explain each layer..."

STEP 4: Scaling (10 minutes)
"To scale to [number] users: [X] servers with [Y] connections
each. Using [message broker] for cross-server communication.
Monitoring [key metrics]."

STEP 5: Deep Dive (10 minutes)
"For [specific aspect]: [detailed explanation of message flow,
failure handling, or optimization]."

STEP 6: Trade-offs (3 minutes)
"Key trade-offs: [discuss alternatives and why you chose
your approach]."
```

Remember

- **No perfect solution** - only trade-offs based on requirements
- **Start simple** - don't over-engineer for small scale
- **Think operations** - monitoring, deployment, failures
- **Justify choices** - explain WHY, not just WHAT
- **Know alternatives** - shows depth of understanding

Quick Technology Decision Guide

```
Question Flowchart:

"Do you need real-time updates?"
├─ No → Regular HTTP API
└─ Yes ↓

"Is it bidirectional (both ways)?"
├─ Yes → WebSocket (or Long Polling if simpler OK)
└─ No ↓
```

```
"How often are updates?"
├─ Every few seconds → SSE or Long Polling
├─ Sub-second → WebSocket (even if unidirectional)
└─ Minutes → Short Polling

"Is it media (audio/video)?"
└─ Yes → WebRTC (peer-to-peer)

This covers 90% of interview scenarios!
```

Good luck with your system design interviews!

Additional Resources

Key Concepts to Master

- WebSocket protocol and handshake
- Difference between polling types
- Stateful vs stateless scaling
- Message broker patterns (Pub/Sub)
- Connection management strategies
- Heartbeat and keepalive mechanisms
- Reconnection strategies
- Load balancing for persistent connections

Further Reading

- RFC 6455 (WebSocket Protocol)
- Server-Sent Events Specification
- WebRTC Architecture Overview
- Real-time System Design Patterns
- Scaling WebSocket Connections
- Load Balancing Stateful Connections

Practice Questions

1. Design a chat application for 1M users
2. Design a live sports score system
3. Design a collaborative whiteboard
4. Design a multiplayer game backend
5. Design a real-time stock dashboard
6. Design a live notification system

For each, consider:

- Technology choice and why
- Scaling approach
- Message flow

- Failure handling
- Monitoring strategy