HTTP/HTTPS

FTP/SFTP

AMQP

MQTT

# Real time communication

## Considerations for Selection:

**Latency Requirements**: WebSocket, WebRTC, or UDP-based protocols for ultra-low latency.

**Scalability**: MQTT or gRPC for large-scale systems.

**Network Conditions**: MQTT or CoAP for unreliable or low-bandwidth networks.

**Data Type**: WebRTC for media, WebSocket/gRPC for general data, SSE for simple updates.

**Ease of Use**: SSE or WebSocket for quick browser-based implementations.

## WebRTC

enables real-time, **peer-to-peer** communication of audio, video, and data directly between clients (**typically browsers or applications**) without requiring an intermediary server for data transfer. It leverages a combination of standardized protocols to achieve low-latency communication, primarily using **UDP** for transport.

**Why is WebRTC Fast?**

WebRTC is designed for low-latency communication, making it fast for real-time applications. Here’s why:

1. **UDP**-Based Transport:  
   WebRTC primarily uses UDP (User Datagram Protocol) instead of TCP. UDP is faster because it avoids TCP’s overhead, such as guaranteed delivery, retransmissions, and congestion control, which add latency.  
   For real-time applications like video calls, dropping a few packets (e.g., a frame of video) is preferable to delaying the stream for retransmission.
2. **Direct Peer-to-Peer** Communication: WebRTC establishes direct connections between peers, reducing the need for data to pass through intermediate servers. This minimizes latency compared to server-relayed protocols like WebSocket or HTTP-based streaming.
3. **Optimized Codecs**: WebRTC uses efficient codecs like VP8, VP9, or H.264 for video and Opus for audio, which are optimized for real-time compression and decompression, balancing quality and speed.  
   These codecs adapt to network conditions dynamically, ensuring smooth performance.
4. **Interactive Connectivity Establishment** (ICE): WebRTC uses ICE to find the optimal path between peers, navigating NATs and firewalls efficiently. It tries multiple connection methods (e.g., direct, STUN, TURN) to establish the fastest route.
5. **Adaptive Bitrate Control**: WebRTC dynamically adjusts bitrate based on network conditions, reducing buffering and ensuring smooth real-time delivery.
6. **Hardware Acceleration**: Many WebRTC implementations leverage hardware acceleration for encoding/decoding media, further reducing processing time.

**Communication structure**

Structure of Request and Response in WebRTCWebRTC does not follow a traditional request-response model like HTTP. Instead, it uses a signaling process to establish a peer-to-peer connection, followed by direct data exchange. Here’s how it works:

1. Signaling Phase:
   * Purpose: Establishes the connection by exchanging metadata (not media/data) between peers.
   * Mechanism:
     + Peers use a separate channel (e.g., WebSocket, HTTP, or a custom server) to exchange Session Description Protocol (SDP) messages and ICE candidates.
     + SDP: Contains metadata about the media (e.g., codecs, formats) and connection details. Example:

v=0

o=- 123456789 2 IN IP4 127.0.0.1

s=-

t=0 0

m=audio 49170 RTP/AVP 0

...

* + - ICE Candidates: Describe possible network paths (IP addresses, ports) for the connection.
  + Process:
    1. Peer A creates an offer (SDP) and sends it via the signaling server.
    2. Peer B receives the offer, generates an answer (SDP), and sends it back.
    3. Both peers exchange ICE candidates to discover the best connection path.

1. Connection Establishment:
   * Once signaling is complete, WebRTC uses ICE to negotiate the connection, attempting:
     + Direct P2P: If peers are on the same network or NAT traversal succeeds.
     + STUN: To map public IPs for NAT traversal.
     + TURN: A relay server as a fallback if direct connection fails (adds latency).
   * The result is a direct RTCPeerConnection for media and/or data.
2. Data Exchange:
   * After connection, peers exchange media (via RTP/RTCP) or data (via DataChannel).
     + RTP (Real-time Transport Protocol): Streams audio/video packets over UDP.
     + RTCP (RTP Control Protocol): Monitors transmission quality and provides feedback.
     + DataChannel: A bidirectional channel for arbitrary data (e.g., text, binary), built on SCTP (Stream Control Transmission Protocol) over DTLS (Datagram Transport Layer Security).
3. No Traditional Request-Response:
   * Unlike HTTP, WebRTC streams data continuously once connected. There’s no discrete request-response cycle; it’s a persistent, real-time flow.

**Where WebRTC is a Better Choice**

WebRTC excels in scenarios requiring low-latency, peer-to-peer communication, particularly for media. It’s preferable over other protocols in the following cases:

1. Video and Audio Conferencing:
   * Why WebRTC?: Native support for real-time audio/video, adaptive codecs, and peer-to-peer connections reduce latency and server costs compared to WebSocket or RTMP.
   * Example: Zoom-like applications, Google Meet.
   * Compared to Others: WebSocket requires server relaying, increasing latency; RTMP is less flexible and outdated for browser use.
2. Live Streaming with Low Latency:
   * Why WebRTC?: Sub-second latency for live video streaming (e.g., <500ms), ideal for interactive streaming.
   * Example: Live gaming streams (Twitch-like), real-time auctions.
   * Compared to Others: HLS (HTTP Live Streaming) has 5-30s latency; RTMP is similar but less browser-friendly.
3. Peer-to-Peer File Sharing:
   * Why WebRTC?: DataChannel allows direct, secure data transfer without server storage, unlike HTTP-based file transfers.
   * Example: Peer-to-peer file-sharing apps like ShareIt.
   * Compared to Others: MQTT or gRPC require server infrastructure; WebSocket is less optimized for large binary transfers.
4. Real-Time Gaming:
   * Why WebRTC?: Low-latency DataChannel for game state synchronization, combined with video/audio for in-game communication.
   * Example: Multiplayer browser games.
   * Compared to Others: UDP-based custom protocols are similar but harder to implement; WebSocket adds server latency.
5. IoT and Telemetry (Niche):
   * Why WebRTC?: DataChannel for real-time sensor data in specific IoT scenarios where MQTT’s broker overhead is undesirable.
   * Example: Real-time drone control.
   * Compared to Others: MQTT is better for constrained devices, but WebRTC suits high-bandwidth, low-latency IoT.

**Best Practices** in WebRTC Usage

1. Use a Reliable Signaling Server:
   * Implement a robust signaling mechanism (e.g., WebSocket or HTTP) to exchange SDP and ICE candidates. Ensure it’s secure (e.g., WSS) and scalable.
2. Optimize ICE Configuration:
   * Include multiple STUN servers for NAT traversal and a TURN server as a fallback (e.g., Google’s STUN: stun.l.google.com:19302).
   * Prioritize direct P2P connections to minimize latency.
3. Handle Network Variability:
   * Implement adaptive bitrate control to adjust media quality based on bandwidth.
   * Use RTCP feedback to monitor and respond to packet loss or jitter.
4. Secure Connections:
   * WebRTC mandates DTLS for encryption and SRTP for secure media. Ensure certificates are properly configured.
   * Validate signaling data to prevent injection attacks.
5. Error Handling:
   * Handle ICE failures, connection drops, and DataChannel errors gracefully with retries or fallback mechanisms.
   * Log errors for debugging (e.g., ICE candidate failures).
6. Optimize for Performance:
   * Use hardware-accelerated codecs (e.g., VP8/VP9) where available.
   * Limit DataChannel message sizes to avoid fragmentation (e.g., <16KB for SCTP).
7. Cross-Browser Compatibility:
   * Test across browsers (Chrome, Firefox, Safari) as WebRTC implementations vary slightly.
   * Use libraries like Adapter.js for JavaScript-based apps to normalize APIs.
8. Scalability with SFU/MCU:
   * For multi-party calls, use a Selective Forwarding Unit (SFU) or Multipoint Control Unit (MCU) to manage streams efficiently, as pure P2P doesn’t scale well for large groups.

**Common Mistakes and Solutions**

1. Mistake: Improper Signaling Setup:
   * Problem: Failing to exchange SDP/ICE candidates correctly leads to connection failures.
   * Solution: Use a reliable signaling server (e.g., WebSocket with WSS). Verify SDP and ICE candidate exchange order (offer → answer → ICE candidates).
2. Mistake: Ignoring TURN Fallback:
   * Problem: Connections fail in strict NAT/firewall environments without a TURN server.
   * Solution: Configure a TURN server (e.g., open-source Coturn) and include it in the ICE configuration.
3. Mistake: Overloading DataChannel:
   * Problem: Sending large or frequent messages causes congestion or dropped messages.
   * Solution: Use smaller messages (<16KB), implement backpressure handling, or switch to media streams for large data.
4. Mistake: Poor Codec Selection:
   * Problem: Choosing incompatible or inefficient codecs leads to high latency or poor quality.
   * Solution: Prefer VP8/VP9 for video and Opus for audio. Test codec compatibility between peers.
5. Mistake: Not Handling Connection State Changes:
   * Problem: Failing to monitor RTCPeerConnection state (e.g., disconnected, failed) causes silent failures.
   * Solution: Listen to onconnectionstatechange and implement reconnection logic.
6. Mistake: Inadequate Error Logging:
   * Problem: Debugging connection issues is hard without detailed logs.
   * Solution: Log ICE candidate errors, SDP mismatches, and DataChannel states. Use tools like Chrome’s webrtc-internals for diagnostics.
7. Mistake: Assuming Universal Browser Support:
   * Problem: Some browsers (e.g., older Safari versions) have partial WebRTC support.
   * Solution: Test across target browsers and use fallback protocols (e.g., WebSocket) for unsupported environments.

RTMP

HLS

DASH

gRPC

WebSocket

CoAP