VIDEO CHAT APPLICATION - COMPLETE WORKFLOW GUIDE

This document explains how your video chat application works from start to finish. After reading this, you'll understand how all the pieces connect and work together.

TABLE OF CONTENTS

---

1. HIGH-LEVEL ARCHITECTURE

Your application has THREE main parts:

```
┌─────────────────┐      ┌─────────────────┐      ┌─────────────────┐
│  Frontend 1     │◄────►│  Signaling      │◄────►│  Frontend 2     │
│  (User A)       │      │  Server         │      │  (User B)       │
│                 │      │  (Node.js)      │      │                 │
└─────────────────┘      └─────────────────┘      └─────────────────┘
     │                                                   │
     └───────────────────────────────────────────────────┘
              Direct WebRTC Connection
              (Video/Audio Streams)
```

Frontend (Next.js + React):

- User interface (buttons, video displays, chat)
- Manages user's camera and microphone
- Handles WebRTC peer connections
- Communicates with signaling server via Socket.IO

Signaling Server (Node.js + Socket.IO):

- Helps users find each other (room management)
- Relays connection information between peers
- Manages chat messages
- Provides TURN/STUN servers for NAT traversal
- Does NOT handle actual video/audio (that's peer-to-peer)

WebRTC Connection:

- Direct peer-to-peer connection for video/audio
- Low latency (no server in the middle)
- Uses TURN servers if direct connection fails

---

2. CORE CONCEPTS YOU NEED TO KNOW

What is Socket.IO?

Socket.IO enables real-time, bidirectional communication between browser and server. Think of it like a phone line that stays open, allowing instant messages in both directions.

Example:

```
User A: "Hey server, tell User B I want to call them"
Server: "Sure! Hey User B, User A wants to call you"
```

What is WebRTC?

WebRTC (Web Real-Time Communication) allows browsers to send video/audio directly to each other WITHOUT going through a server. This is much faster and more efficient.

Why it's important:

- Video/audio streams are huge amounts of data
- Sending through a server would be slow and expensive
- Peer-to-peer is faster and reduces server costs

What is SDP (Session Description Protocol)?

SDP is like a "business card" that describes what media you can send/receive. It contains:

- What codecs you support (H.264, VP8, Opus, etc.)
- What media you want (audio, video, or both)
- Network information

The Offer/Answer Exchange:

1. User A creates an "offer" (their business card)
2. User A sends it to User B via signaling server
3. User B creates an "answer" (their business card)

4. User B sends it back to User A
5. Now both know how to communicate!

What are ICE Candidates?

ICE candidates are network addresses where you can be reached. Your browser finds multiple ways it might be reachable:

- Local network address (192.168.x.x)
- Public IP address
- TURN server relay address

Both peers exchange ALL their candidates and try each one until they find one that works.

What are TURN/STUN Servers?

STUN (Session Traversal Utilities for NAT):

- Helps you discover your public IP address
- Like asking "what's my phone number?" to someone outside your office

TURN (Traversal Using Relays around NAT):

- Relay server when direct connection fails
- Like having a secretary relay messages when you can't talk directly
- More expensive (uses server bandwidth) but always works

---

3. APPLICATION STARTUP FLOW

Step 1: App Loads

When your Next.js app starts:

1. **SocketProvider component mounts**
    - Located in `socket.tsx`
    - Wraps your entire app
    - useEffect hook runs (runs once when component loads)
2. **Socket connection established**

```
Frontend                    Server
   |                          |
   |---- Connect Request ----->|
   |<--- Connection ID --------|
   |                          |
```

3. **PeerService initializes**
    - Creates RTCPeerConnection (but doesn't activate it yet)
    - Fetches TURN/STUN servers from Twilio (via your backend)

Step 2: User Interface Appears

- User sees room input and join button
- Camera/microphone are NOT accessed yet (privacy!)
- Socket connection indicator shows "connected"

Why useEffect here? useEffect runs after the component renders. We need this because:

- Socket.IO only works in browser (not during server-side rendering)
- We want to connect exactly once when app starts
- We need to clean up the connection when app closes

---

4. JOINING A ROOM FLOW

Step 1: User Enters Room Code

User types room ID (e.g., "room123") and clicks "Join"

Step 2: Request Media Permissions

javascript

```javascript
navigator.mediaDevices.getUserMedia({ video: true, audio: true })
```

- Browser shows permission popup
- User grants access to camera/microphone
- App receives MediaStream object

Step 3: Join Room via Socket

```
Frontend                    Server
   |                          |
   |-- room:join ------------>|
   |   {room: "123",          |
   |    userName: "Alice"}    |
   |                          |
   |                    [Check room size]
   |                    [If full, reject]
   |                    [If ok, add to room]
   |                          |
   |<-- room:joined -----------|
   |   {users: [{id, name}]}  |
   |                          |
```

Step 4: Server Response

If Room is Full:

```
Server sends: "room:full" event
Frontend shows: Error message
User action: Try different room
```

If Room has Space:

```
Server sends: "room:joined" with list of all users
Frontend: Shows waiting room or starts call if 2nd user
```

Step 5: Notify Other User

If someone is already in the room:

```
Server broadcasts: "user:joined" to existing users
Other user receives: New user info
Other user can: Initiate call or wait
```

---

5. MAKING A CALL FLOW

This is the most complex part - where WebRTC magic happens!

Step 1: Initiating the Call

User A clicks "Call" button (or auto-calls when User B joins)

Code flow:

javascript

```javascript
// 1. Add local stream to peer connection
await peerService.addLocalStream(myStream);

// 2. Create offer
const offer = await peerService.getOffer();

// 3. Send offer via socket
socket.emit("user:call", { to: userB_id, offer });
```

Step 2: The Offer Travels

```
User A              Server              User B
  |                   |                   |
  |-- user:call -------->|                   |
  |    (offer SDP)       |                   |
  |                   |-- incoming:call ---->|
  |                   |    (offer SDP)       |
```

Step 3: User B Receives Call

User B's frontend receives "incoming:call" event

Code flow:

javascript

```javascript
socket.on("incoming:call", async ({ from, offer }) => {
  // 1. Add User B's local stream
  await peerService.addLocalStream(myStream);

  // 2. Set remote offer (User A's info)
  // 3. Create answer
  const answer = await peerService.getAnswer(offer);

  // 4. Send answer back
  socket.emit("call:accepted", { to: from, ans: answer });
});
```

Step 4: The Answer Returns

```
User B              Server              User A
  |                   |                   |
  |-- call:accepted ---->|                   |
  |   (answer SDP)     |                   |
  |                   |-- call:accepted ---->|
  |                   |   (answer SDP)     |
```

Step 5: User A Sets Remote Answer

javascript

```
socket.on("call:accepted", async ({ ans }) => {
  // Set User B's answer
  await peerService.setRemoteAnswer(ans);

  // Now both peers know each other's capabilities!
});
```

Step 6: ICE Candidate Exchange

While all this is happening, BOTH peers are finding network addresses:

```
User A              Server              User B
  |                    |                    |
  |-- ice:candidate ---->|-- ice:candidate ---->|
  |                    |                    |
  |<-- ice:candidate ----|<-- ice:candidate ----|
  |                    |                    |
  |-- ice:candidate ---->|-- ice:candidate ---->|
  |                    |                    |
  (continues until connection established)
```

What's happening:

1. Each peer's browser finds possible network addresses
2. Each candidate is immediately sent to the other peer
3. Both peers try connecting using each candidate pair
4. First successful connection wins!

Step 7: Connection Established!

```
User A ←──────────────────────────────────────→ User B
       Direct WebRTC Connection Established
       Video and Audio flowing peer-to-peer
```

Events that fire:

- ICE connection state: "connected"
- Connection state: "connected"

- "track" event fires on both peers (receiving remote video/audio)

Step 8: Display Remote Video

javascript

```javascript
peerService.onTrack((event) => {
  // event.streams[0] contains remote video/audio
  remoteVideoElement.srcObject = event.streams[0];
});
```

---

6. RECEIVING A CALL FLOW

From User B's perspective (simplified):

1. **Receive "incoming:call" event**
   - Play ringing sound
   - Show "User A is calling" notification
2. **User clicks "Accept"**
   - Access camera/microphone
   - Add local stream to peer connection
   - Generate answer from received offer
   - Send answer back
3. **ICE candidates exchange**
   - Automatically handled by peer connection
   - Each candidate sent via socket
4. **Connection established**
   - Remote video appears
   - Audio starts flowing
   - Chat becomes active

---

7. CHAT MESSAGES FLOW

Chat works independently from video/audio (uses Socket.IO, not WebRTC):

Sending a Message

```
User A                    Server                  User B
  |                         |                         |
  |-- chat:message -------->|                         |
  |  {room, message, name}  |                         |
  |                         |                         |
  |                   [Store in memory]               |
  |                   [Add timestamp]                 |
  |                         |                         |
  |<-- chat:message --------|-- chat:message -------->|
  |  {id, sender, text}     |  {id, sender, text}     |
  |                         |                         |
```

Key points:

- Messages go through server (not peer-to-peer)
- Server stores messages in Map (lost on server restart)
- Everyone in room receives the message (including sender)
- Messages include: ID, sender name, text, timestamp

Why not use WebRTC data channel?

- Socket.IO is already connected
- Simpler implementation
- Works even if WebRTC fails
- Messages need to go through server for storage anyway

---

8. LEAVING/ENDING CALL FLOW

User Clicks "End Call" or "Leave Room"

Step 1: Notify Other Peer

javascript

```javascript
socket.emit("call:end", { to: otherUserId });
// or
socket.emit("leave:room");
```

Step 2: Clean Up Peer Connection

javascript

```javascript
await peerService.reset();
// This:
// - Removes all event listeners
// - Closes peer connection
// - Clears senders array
// - Creates fresh peer for next call
```

Step 3: Stop Media Tracks

javascript

```javascript
myStream.getTracks().forEach(track => track.stop());
// Turns off camera and microphone
// Camera light turns off
```

Step 4: Update UI

```
- Hide remote video
- Show "Call ended" message
- Return to waiting room or lobby
- Clear chat (optional)
```

Server-Side Cleanup

When user disconnects:

javascript

```javascript
socket.on('disconnect', () => {
  // Get user's room
  // Notify others: "user:left"
  // Remove from room
  // Log updated room count
});
```

Room Management:

- Room automatically deleted when empty
- Messages cleared (in current implementation)
- Next users joining create fresh room

---

9. KEY COMPONENTS DEEP DIVE

PeerService (Singleton Pattern)

Why singleton?

javascript

```javascript
export default new PeerService();
```

- Only ONE peer connection per user
- Shared across all components
- Prevents creating multiple connections

Critical Methods:

`createPeer()`

- Called once on startup
- Fetches TURN/STUN servers
- Creates RTCPeerConnection
- Sets up basic event listeners

`getOffer()` / `getAnswer()`

- Generate SDP descriptions
- Automatically set as local description
- Must be called in correct order

`addLocalStream()`

- Intelligently handles track replacement
- Checks if sender exists for audio/video
- Replaces track if exists (avoids renegotiation)
- Adds new track if doesn't exist

`reset()`

- Essential for starting new calls
- Removes ALL event listeners (prevents memory leaks)
- Closes old connection properly
- Creates fresh connection

SocketService (Singleton Pattern)

Connection Management:

javascript

```
connect() {
  if (this.socket?.connected) return this.socket; // Don't reconnect
  if (this.socket && !this.socket.connected) {
    this.socket.connect(); // Reconnect existing
  }
  // Otherwise create new
}
```

Auto-Reconnection:

- Tries 10 times automatically
- Waits 1-5 seconds between attempts
- Resets counter on successful connection

Transport Priority:

javascript

```
transports: ['polling', 'websocket']
```

1. Starts with polling (HTTP long-polling) - most reliable
2. Upgrades to websocket if possible - faster
3. Falls back to polling if websocket fails

SocketProvider (React Context)

Why Context?

- Avoids "prop drilling" (passing socket through every component)
- Components anywhere can use: `const { socket } = useSocket();`
- Single source of truth for connection status

State Management:

javascript

```
const [socket, setSocket] = useState(null);
const [connectionStatus, setConnectionStatus] = useState("connecting");
const [retryCount, setRetryCount] = useState(0);
```

- `socket`: The actual connection
- `connectionStatus`: For UI feedback
- `retryCount`: Shows connection attempts

Why useEffect?

javascript

```javascript
useEffect(() => {
  // Setup code
  return () => {
    // Cleanup code
  };
}, []); // Empty array = run once on mount
```

- Runs after component renders
- Only runs once (empty dependency array)
- Cleanup function runs when component unmounts
- Perfect for subscriptions and connections

---

10. COMMON QUESTIONS ANSWERED

Q: Why do we need both Socket.IO and WebRTC?

Socket.IO:

- Signaling (exchanging connection info)
- Room management
- Chat messages
- Server-relayed communication

WebRTC:

- Actual video/audio streaming
- Peer-to-peer (no server)
- Low latency
- High bandwidth

Think of it like:

- Socket.IO = Phone number exchange
- WebRTC = The actual phone call

Q: What happens if network fails during call?

ICE Connection States:

```
connected → disconnected → [waiting] → failed
```

1. Connection becomes "disconnected"
2. ICE tries to reconnect automatically
3. Gathers new candidates
4. If reconnection fails → "failed" state
5. Your code can detect this and restart call

Q: Why fetch ICE servers instead of hardcoding?

Twilio provides temporary credentials:

- TURN servers need authentication (prevent abuse)
- Credentials expire (security)
- Dynamic server selection (performance)
- Failover options (reliability)

Q: Can more than 2 people join a room?

Currently: **No** - enforced by server:

javascript

```javascript
if (clientCount >= 2) {
  socket.emit("room:full");
  return;
}
```

To support 3+ people:

- Mesh topology (everyone connects to everyone)
- SFU (Selective Forwarding Unit) server
- MCU (Multipoint Control Unit) server

Each approach has trade-offs in complexity and performance.

Q: What happens to old messages when server restarts?

Current implementation:

javascript

```javascript
const roomMessages = new Map();
```

- Messages stored in memory
- Lost on server restart
- Lost when room becomes empty

To persist messages:

- Use database (MongoDB, PostgreSQL)
- Use Redis (in-memory database with persistence)
- Send message history when user joins

Q: Why can't we store state in localStorage in artifacts?

localStorage is a browser API that's not available in the Claude.ai artifact environment. Instead:

- React components use `useState` / `useReducer`
- HTML artifacts use JavaScript variables
- All data stays in memory during session

Q: How does peer negotiation work?

When needed:

- Adding/removing audio or video track
- Changing video resolution
- Network conditions change

Process:

1. Peer A: `createOffer()` → send via "peer:nego:needed"
2. Peer B: receive offer → `createAnswer()` → send via "peer:nego:done"
3. Peer A: receive answer → connection renegotiated!

Q: What's the difference between signalingState, iceConnectionState, and connectionState?

signalingState:

- Where in offer/answer exchange
- Values: stable, have-local-offer, have-remote-offer, have-local-pranswer, have-remote-pranswer

iceConnectionState:

- Status of ICE candidate gathering and checking
- Values: new, checking, connected, completed, failed, disconnected, closed

connectionState:

- Overall connection status
- Values: new, connecting, connected, disconnected, failed, closed

- Most useful for UI feedback

COMPLETE CALL SEQUENCE DIAGRAM

```
USER A                SERVER                USER B
  |                     |                     |
  |--[Join room]---------->|                     |
  |<--[Room joined]---------|                     |
  |                     |<--[Join room]-------------|
  |<--[User joined]---------|--[Room joined]---------->|
  |                     |--[User joined]---------->|
  |                     |                     |
  |--[user:call + offer]--->|                     |
  |                     |--[incoming:call + offer]->|
  |                     |                     |
  |--[ice:candidate]------->|--[ice:candidate]-------->|
  |<--[ice:candidate]-------|<--[ice:candidate]---------|
  |                     |                     |
  |                     |<--[call:accepted + ans]---|
  |<--[call:accepted + ans]-|                     |
  |                     |                     |
  |--[ice:candidate]------->|--[ice:candidate]-------->|
  |<--[ice:candidate]-------|<--[ice:candidate]---------|
  |                     |                     |
  |←══════════════════════════════════════════════════════════→|
       WebRTC Connection Established (Video/Audio)
  |                     |                     |
  |--[chat:message]-------->|--[chat:message]--------->|
  |<--[chat:message]--------|<--[chat:message]---------|
  |                     |                     |
  |--[call:end]------------>|--[call:ended]----------->|
  |                     |                     |
```

SUMMARY

Your video chat app works through a carefully choreographed dance:

1. **Startup**: Connect to signaling server, initialize peer connection

2. **Join**: Enter room, request media, notify others
3. **Call**: Exchange SDP offers/answers, exchange ICE candidates
4. **Connect**: Establish direct WebRTC connection
5. **Communicate**: Stream video/audio peer-to-peer, send chat via server
6. **End**: Clean up connections, stop media, notify other user

The beauty is in the separation:

- **Signaling** (Socket.IO): Light coordination traffic
- **Media** (WebRTC): Heavy video/audio traffic peer-to-peer

This architecture is scalable, efficient, and follows industry best practices!

NEXT STEPS TO IMPROVE

1. **Add error handling UI**: Show friendly messages when things fail
2. **Persist chat**: Use database instead of in-memory Map
3. **Add screen sharing**: WebRTC supports this!
4. **Better reconnection**: Auto-restart call if connection drops
5. **Add audio/video toggles**: Mute/unmute functionality
6. **Room history**: Show who was in room previously
7. **Multiple rooms**: Let users switch between rooms
8. **Mobile optimization**: Test on phones, adjust UI

You now understand your entire video chat application! 🎉