PeerPlay Quiz

Synchronized quiz fun with friends, enhanced by direct P2P video connection.

Team Members

R. Amogh, Rahul Anand, Mohith Velivela, Abhinay, Vignesh

April 18, 2025

Contents

1 Motivation and Goals

We recognized that while many tools exist for remote communication and online games, few capture the lively, interactive dynamic of playing trivia together in the same room. We were motivated by the shared desire to bridge this gap. Our goal with PeerPlay Quiz was to combine our skills to build a platform that offers a perfectly synchronized quiz experience using WebSockets, while crucially adding that missing social layer through optional, integrated P2P video chat, allowing friends to truly connect and share the fun live, no matter the distance.

2 Technical Stack

- Frontend: Next.js (React, TypeScript), TailwindCSS, Framer Motion
- Communication: Socket.io (WebSocket), WebRTC for P2P video
- Backend: Node.js, Express, Socket.io server
- Tools: npm, ESLint, Prettier
- Deployment: (e.g., Vercel / Heroku)

3 Features Implemented

- Real-time multiplayer quiz sessions with synchronized timers
- Lobby system for session creation and joining
- Adaptive question timing based on network conditions
- Instant feedback and animated result screens
- Dark mode and responsive design
- Integrated P2P video chat in conference mode
- Network performance monitoring (latency, packet loss)

4 Getting Started

4.1 Prerequisites

- Node.js (v18+) and npm (v9+)
- A modern web browser

4.2 Setup and Run

1. Clone the repository: git clone https://github.com/Amogh-2404/SocketQuiz cd quiz-show/ (or cd SocketQuiz if you cloned the repository)
2. Install dependencies: npm install
3. Start the server (in separate terminal): cd server npm install npm start
4. Run the frontend: cd .. npm run dev
5. Open http://localhost:3000 in your browser.

5 Project Structure

The repository structure is as follows:

- quiz-show/: Main application directory
  - .gitignore: Git ignore rules
  - package.json: Frontend dependencies and scripts
  - next.config.js: Next.js configuration
  - tsconfig.json: TypeScript configuration
  - tailwind.config.js: Tailwind CSS configuration
  - public/: Static files (images, icons)
  - src/: Application source code
    - app/: Next.js pages and components
  - server/: Backend server code
    - data/: Quiz question data files
    - index.js: Express and Socket.io server entrypoint
    - networkMonitor.js: Network monitoring module
    - test-server.js: Test server scripts

6 Detailed Implementation

6.1 Synchronization Mechanism

To achieve precise synchronization across clients, we implement a high-resolution ping–pong handshake and drift compensation:

- Client records a timestamp via `performance.now()` and emits `'ping'` with `clientTime`.
- Server responds immediately with `'pong'` echoing the received timestamp and its own `serverTime`.
- Client computes round-trip time (RTT) as `performance.now() - clientTime` and estimates one-way latency = `RTT / 2`.
- When broadcasting `'newQuestion'`, server includes `timestamp: Date.now()`. Clients schedule UI update with `setTimeout(serverTimestamp + latencyEstimate - Date.now())`, achieving typical skew <50 ms.
- To correct long-term drift, a re-sync occurs every 5 questions: clients repeat ping/pong and adjust future timers.

Theory:

This timestamp-based synchronization parallels the Network Time Protocol (NTP) model. By measuring half the round-trip delay via ping–pong and scheduling events with latency compensation, clients achieve clock alignment within tens of milliseconds. Periodic re-sync mitigates local clock drift and network jitter, ensuring consistency across distributed peers.

GameContext.tsx snippet: const start = performance.now(); socket.emit('ping', start); socket.on('pong', (start: sent) => const now = performance.now(); const rtt = now - sent; setLatency(rtt / 2););

6.2 Socket.io Implementation Details

We leverage Socket.io for event-driven, room-based messaging:

- On connecting, client emits `socket.emit('join', roomId, playerName )`. Server calls `socket.join(roomId)` and broadcasts the new player list.
- Clients emit `'answerSubmit'` with payload { questionId, choice }. Server validates, updates scores, then emits `'questionResult'` to the room.
- Disconnections are handled by `socket.on('disconnect')`, removing players and notifying remaining clients.

Server (index.js): io.on('connection', socket => socket.on('join', (roomId,name) => socket.join(roomId); io.to(roomId).emit('playerList', getPlayers(roomId));); socket.on('answerSubmit', data => handleAnswer(socket, data)););

6.3 WebRTC Connection Flow

Conference mode uses browser-native WebRTC:

1. Capture media: `navigator.mediaDevices.getUserMedia( video: true, audio: true )`.
2. Create `RTCPeerConnection` with STUN servers for NAT traversal.
3. Client A creates an SDP offer, `pc.setLocalDescription(offer)`, emits via Socket.io `'signal'` event.
4. Client B on `'signal'` sets remote description, creates and returns an answer.
5. ICE candidates are exchanged through the same channel until connectivity is established.
6. Streams are attached to video elements in `VideoGrid.tsx`.

Theory:

WebRTC uses Session Description Protocol (SDP) for media negotiation and ICE for NAT traversal. STUN servers discover public-facing addresses, and trickle ICE speeds up connection by emitting candidates as they arrive. This lowers setup latency and maximizes connectivity success without a TURN relay.

Signaling snippet: pc.createOffer().then(offer => pc.setLocalDescription(offer); socket.emit('signal', to: peerId, sdp: offer);); socket.on('signal', (from, sdp, candidate) => ...);

6.4 Conference Mode (P2P Video Chat) Internals

In conference mode, the client builds a full peer-to-peer mesh for video/audio streams without relaying media through the server:

- **Media Capture**: uses `navigator.mediaDevices.getUserMedia( video, audio )` in `VideoGrid.tsx`, with toggles for camera/mic.
- **Signaling Channel**: leverages Socket.io events: `video-peer-join`, `video-offer`, `video-answer`, and `video-ice-candidate` to exchange SDP and ICE.
- **Peer Connections**: instantiates `new SimplePeer( initiator, trickle: true, stream: localStream, config: { iceServers: [{ urls: 'stun:stun.l.google.com:19302' }], bundlePolicy: 'max-bundle', rtcpMuxPolicy: 'require' } )`.

- **ICE Trickle**: enables incremental ICE candidate delivery, speeding NAT traversal and reducing setup latency.
- **Direct P2P Mesh**: once signaling completes, media flows directly client-to-client, minimizing server bandwidth and latency.
- **Resource Cleanup**: the `cleanup()` function stops media tracks and closes peer connections on unmount or mode change, preventing leaks.
- **Fallback**: if SimplePeer is unavailable, raw `RTCPeerConnection` logic in `VideoGrid.tsx` mirrors signaling and track handling.

Theory:

A full-mesh P2P topology incurs O(n^2)$mediastreams, suitableforsmallgroups(< 6).Iteliminatesserverbandwi$
$to - endlatency, leveragingdirectUDPtransportandbrowser - nativeDTLS/SRTPsecurity.$

Key Code Snippet (VideoGrid.tsx): const peer = new SimplePeer(initiator, trickle: true, stream: localStream, config: iceServers: [urls: 'stun:stun.l.google.com:19302'], bundlePolicy: 'max-bundle', rtcpMuxPolicy: 'require'); peer.on('signal', data => const event = data.type === 'offer' ? 'video-offer' : data.type === 'answer' ? 'video-answer' : 'video-ice-candidate'; socket.emit(event, to: peerId, from: myId, sdp: data, candidate: data.candidate);); peer.on('stream', stream => // attach stream to <video>);

6.5 Library Selection & Alternatives

We evaluated alternatives for performance and developer ergonomics:

- **Socket.io** vs pure `ws`: chosen for auto-reconnect, fallbacks, and built-in rooms.
- **WebRTC P2P** vs SFU/MCU: P2P mesh avoids server media costs, sufficient for small groups.
- **Next.js** vs CRA: supports SSR, static export, and integrated API routes for future backend additions.
- **TailwindCSS** & **Framer Motion**: Utility-first styling plus declarative animations accelerates UI development.

6.6 Network Monitoring & Adaptive Timing

We continuously monitor latency and packet loss to adjust gameplay: useEffect(() => const id = setInterval(() => const t0 = performance.now(); socket.emit('ping', t0); , 5000); return () => clearInterval(id); , []); Packet loss = $(sent - received)/sent$. If `latency > 300ms` or loss > 10

Theory:

Monitoring uses a sliding window of recent pings to estimate jitter and packet loss. Adjusting timers based on average latency aligns with adaptive buffering techniques in multimedia, balancing responsiveness and tolerance for network variability.

6.7 Session Management & Scalability

The server maintains game sessions in a `Map` with a maximum of `MAX`$_S ESSIONSconcurrentlobbies.SessionIDsa$
$functiongenerateSessionId()returnMath.random().toString(36).substring(2, 10);Onclient$`'join'`$:$
$socket.on('join', (name, mode) => if(!canCreateNewSession())//mergeorrejectlogicconstsessionId = assi$
$startafter$`LOBBY`$_T IMERsecondsorwhenallplayersemit$`'ready'`$: socket.on('ready', () => markReady(socket$