Real-Time Study Rooms — Architecture & Socket Event Flow

Purpose: concise, developer-focused design doc you can use to implement the app.

---

1. High-Level Overview

A web app where users create/join **Study Rooms** that provide: - Group **video/audio** using **WebRTC** (P2P, later fallback to TURN).
- Real-time **text chat** and **presence** with **Socket.IO**.
- Collaborative **whiteboard** synchronized via Socket.IO.
- File sharing (uploads/links) and room management (mute/kick).

2. Components

- **Client (React)**
- Socket.IO client
- WebRTC peer connections
- UI: room lobby, participant grid, chat panel, whiteboard canvas
- **Backend (Node.js + Express)**
- REST API (rooms, auth, uploads)
- Socket.IO server (namespaces/rooms)
- Signaling channels for WebRTC (offer/answer/ICE)
- Persistence: MongoDB (or PostgreSQL)
- **Media Infrastructure**
- STUN servers (Google STUN or coturn)
- TURN server for NAT traversal under production
- **Storage**
- File uploads -> cloud storage (e.g., Cloudinary / S3)
- DB for users, rooms, messages, whiteboard snapshots
- **Scaling**
- Socket.IO Redis adapter (pub/sub) for multi-instance socket scaling
- Horizontal scaling for Node servers behind load balancer

3. Data Models (MongoDB-like)

```
// User
{ _id, name, email, passwordHash, avatarUrl, createdAt }

// Room
{ _id, code, title, ownerId, participants: [{userId, name, joinedAt}],
createdAt, settings }
```

```
// Message
{ _id, roomId, senderId, text, attachments, timestamp }

// WhiteboardSnapshot
{ _id, roomId, dataUrl, createdAt, savedBy }
```

4. Socket.IO Structure & Namespaces

- Use single namespace or `/rooms` namespace. Example: `/rooms`
- Within namespace use Socket.IO rooms keyed by `roomId` (or `room:CODE`).

Connection flow: 1. Client connects to namespace: `io('/rooms', { auth: { token } })`. 2. Server validates user and joins socket to room: `socket.join(roomId)`.

5. Socket Events (canonical list)

Client -> Server

- `joinRoom` { roomId, name, role } — request to join; server responds with `joined` or `error`.
- `leaveRoom` { roomId }
- `chat:message` { roomId, text, attachments? }
- `whiteboard:draw` { roomId, ops } — small ops (stroke segments).
- `file:uploadMeta` { roomId, fileUrl, name, size }
- `signal:offer` { toSocketId, from, sdp }
- `signal:answer` { toSocketId, from, sdp }
- `signal:ice` { toSocketId, from, candidate }
- `control:mute` { targetSocketId }
- `presence:typing` { roomId, isTyping }

Server -> Client

- `joined` { roomState } — current participants, room settings.
- `user:joined` { socketId, user } — broadcast when someone enters.
- `user:left` { socketId }
- `chat:message` { message }
- `whiteboard:update` { ops }
- `file:shared` { fileMeta }
- `signal:offer` { fromSocketId, sdp }
- `signal:answer` { fromSocketId, sdp }
- `signal:ice` { fromSocketId, candidate }
- `control:muted` { targetSocketId, by }
- `error` { code, message }

Event payload examples

```
// chat:message
{ "roomId":"abc123", "text":"Hey everyone", "senderId":"u1", "timestamp":
168000000 }

// whiteboard:draw (small op)
{ "roomId":"abc123", "ops": [{"type":"stroke","points":[[x,y],
[x,y]],"color":"#000","width":2}] }
```

6. WebRTC Signaling Flow (P2P mesh)

Sequence summary: 1. User A joins room. Server adds to participant list and broadcasts `user:joined`. 2. Existing participants send a `signal:offer` to the new user's socketId via server (server forwards to target). 3. New user replies with `signal:answer` to each existing peer. 4. ICE candidates exchanged via `signal:ice` events.

```
sequenceDiagram
    participant A as Peer A
    participant S as Signaling (Socket.IO)
    participant B as Peer B

    A->>S: joinRoom(roomId)
    S->>B: user:joined(A.socketId)
    B->>S: signal:offer(to=A.socketId, sdp)
    S->>A: signal:offer(from=B.socketId, sdp)
    A->>S: signal:answer(to=B.socketId, sdp)
    S->>B: signal:answer(from=A.socketId, sdp)
    A->>S: signal:ice(candidate)
    S->>B: signal:ice(candidate)
```

Notes: - For N participants this is O(N^2) peers — fine up to ~6–8 peers. For larger rooms consider an SFU (e.g., mediasoup, Jitsi, Janus). - Always attach metadata: `userId`, `displayName`, `role` to signaling messages so peers can present UI properly.

7. Whiteboard Strategy

- Send compact operations (vector strokes) rather than full image blobs.
- Use client-side throttling/debouncing to limit ops frequency (eg. group points every 50ms).
- Persist occasional snapshots (every X minutes or on `save`) to DB or cloud storage.
- On new join, server sends latest snapshot + replay ops since snapshot.

8. File Sharing

- Client uploads to cloud storage via signed URL from backend.
- After upload, client emits `file:uploadMeta` with URL; server broadcasts `file:shared`.

- For large files consider chunked upload or dedicated file service.

9. REST API Endpoints (examples)

- `POST /api/auth/login` — login
- `POST /api/auth/signup`
- `POST /api/rooms` — create room -> returns roomId & joinLink
- `GET /api/rooms/:id` — room metadata
- `POST /api/upload-url` — get signed URL for direct upload
- `GET /api/rooms/:id/messages` — fetch chat history (pagination)

10. Security & Privacy

- Authenticate Socket.IO connections (token in `auth` handshake).
- Authorize actions: only owner can kick/mute, etc.
- Enforce HTTPS + secure cookies for auth.
- Use CSP, sanitize chat content, validate uploads.
- Use TURN server to avoid leaking direct IPs (and consider privacy policy).

11. Scaling & Production Considerations

- Use **Redis adapter** for Socket.IO so events propagate between instances.
- Use **coturn** TURN server (self-host or managed) for reliable media.
- Monitor CPU & network — P2P media uses client's bandwidth; for many participants SFU is necessary.
- Add rate limits for event spam (chat, whiteboard ops).

12. Deployment Diagram (text)

- Client (React) -> Load Balancer -> Node.js container(s) (Socket.IO + REST)
- Node.js -> Redis (pub/sub) -> Node.js instances
- Node.js -> MongoDB Atlas (or managed DB)
- STUN/TURN servers reachable by clients
- File storage: S3/Cloudinary

13. Minimal Sequence to Implement (step-by-step)

1. Scaffold Node.js + Express + Socket.IO server. Add simple REST `POST /rooms`.
2. Build React room lobby and connect to Socket.IO namespace `/rooms`.
3. Implement `joinRoom` flow and server broadcast of `joined`.
4. Implement text chat and presence.
5. Add simple WebRTC signaling (offer/answer/ice) to exchange tracks and render remote streams.
6. Implement whiteboard ops and broadcast mechanism.
7. Add file upload flow and DB persistence for messages.
8. Add auth and persistent history.

14. Sample Socket.IO server pseudocode (core parts)

```
io.of('/rooms').on('connection', (socket) => {
  const user = socket.handshake.auth.user;

  socket.on('joinRoom', async ({ roomId, name }) => {
    // validate
    socket.join(roomId);
    // send room state
    const roomState = await getRoomState(roomId);
    socket.emit('joined', roomState);
    socket.to(roomId).emit('user:joined', { socketId: socket.id, user });
  });

  socket.on('signal:offer', ({ to, sdp }) => {
    io.of('/rooms').to(to).emit('signal:offer', { from: socket.id, sdp });
  });

  socket.on('signal:answer', ({ to, sdp }) => {
    io.of('/rooms').to(to).emit('signal:answer', { from: socket.id, sdp });
  });

  socket.on('signal:ice', ({ to, candidate }) => {
    io.of('/rooms').to(to).emit('signal:ice', { from: socket.id, candidate });
  });

  socket.on('chat:message', (msg) => {
    // save to DB then broadcast
    io.of('/rooms').to(msg.roomId).emit('chat:message', msg);
  });
});
```

15. UI Wireframe (text)

- Left: Participant grid (top) / Video controls (bottom)
- Right: Chat panel (tabs: Chat | Files | Participants)
- Bottom overlay: Whiteboard toggle / Share screen / Raise hand

---

If you want next: I can **generate the server scaffold** (Node.js + Express + Socket.IO) with example endpoints and a minimal signaling implementation, or **create React starter code** (video grid + Socket.IO client + WebRTC connect). Which do you want me to produce now?