# CS544 Honors Project 2 Architecture Doc

The project's called "N of Us Are Lying" https://github.com/the-snesler/n-of-us-are-lying/tree/main. It's a game played by several players on phones, a host on desktop, and a backend that brokers communications between the groups.

This project implements a resilient distributed system using a hybrid client-authority architecture, designed to maintain game state consistency across network partitions while preventing cheating through cryptographic state isolation.

## Technology Stack

**Backend**:

- Bun (JavaScript runtime) - Fast WebSocket server with native TypeScript support
- WebSocket protocol - Bidirectional, real-time communication
- In-memory data structures - Room and player session management

**Frontend**:

- React - Component-based UI framework
- XState - Finite state machine library for deterministic state management
- Web Crypto API - AES-256-GCM encryption for state recovery
- TypeScript - Type-safe message passing and state definitions
- Zod - Runtime schema validation for network messages

## System Architecture Overview

```
┌─────────────────────────────────────────────────────────────┐
│                     Backend (Bun Server)                    │
│                                                             │
│  ┌────────────┐  ┌──────────────┐  ┌──────────────────┐     │
│  │   Room     │  │  WebSocket   │  │   Player Token   │     │
│  │ Management │  │  Routing     │  │  Authentication  │     │
│  └────────────┘  └──────────────┘  └──────────────────┘     │
│                                                             │
└─────────────┬───────────────────────────────┬───────────────┘
              │                               │
          WebSocket                       WebSocket
              │                               │
       ┌──────▼──────────┐             ┌──────▼──────────┐
       │  Host Client    │             │ Player Client   │  (x N)
       │  ┌──────────┐   │             │ ┌───────────┐   │
       │  │ XState   │   │             │ │   View    │   │
       │  │ Machine  │   │             │ │ Projection│   │
       │  │          │   │             │ └───────────┘   │
       │  │ (SSOT)   │   │             │                 │
       │  └──────────┘   │             │ ┌───────────┐   │
       │                 │             │ │ Encrypted │   │
       │  ┌──────────┐   │             │ │ Host State│   │
       │  │ State    │◄──┼─────────────┼─┤ (Backup)  │   │
       │  │Encryption│   │  Recovery   │ └───────────┘   │
       │  └──────────┘   │             └─────────────────┘
       └─────────────────┘
```

```
State Flow:
  Host: State Change → Encrypt → Broadcast to all players → Player UI Update
  Player: User Action → Send to Backend → Route to Host → State Machine Transition

Recovery Flow:
  Host Crash → Reconnect → Request from Random Player → Decrypt → Restore State
```

## System Architecture

The system consists of three distributed components:

1. **Backend Coordination Service** (Bun + WebSockets)
   - Routes messages between host and players without interpreting game logic
   - Manages WebSocket connections and room lifecycle
   - Handles player authentication via secure tokens
2. **Host Client** (React + XState)
   - Authoritative state manager for all game logic
   - Implements deterministic state machine (XState) for consistency
   - Broadcasts state updates to all connected players
   - Encrypts and distributes recovery checkpoints to players
3. **Player Clients** (React)
   - Thin clients receiving state projections from host
   - Store encrypted host state for disaster recovery
   - Send player actions to host via backend relay

Game state resides on the host client rather than the backend server. This reduces backend complexity and latency (direct state updates without server-side validation), while the cryptographic scheme prevents cheating despite client-side authority. It also allows for easy horizontal scaling of the backend, as it remains stateless, and multiple game types can hosted on the same backend infrastructure.

## Fault Tolerance & Recovery Mechanisms

The system implements multiple layers of resilience against network failures:

### Host Disconnection Recovery

When the host crashes or disconnects:

1. On reconnection, host queries backend for connected players
2. If players exist, host randomly selects a player for state recovery
3. Host sends `REQUEST_STATE_RECOVERY` message to selected player
4. Player responds with encrypted state snapshot (encrypted with host's secret token)
5. Host decrypts snapshot using its persisted token and restores XState machine
6. Game continues from the exact state before disconnection

### Player Reconnection

Players maintain persistent sessions across disconnections:

1. Each player receives a unique `reconnectToken` on initial join (stored in `sessionStorage`)

2. On disconnect, backend marks player as disconnected with timestamp
3. Player automatically attempts reconnection with exponential backoff
4. Within 30-second window, player can reconnect using stored `playerId` and `reconnectToken`
5. Host receives `PLAYER_CONNECTED` event and marks player as active again
6. Player receives latest game state snapshot and rejoins seamlessly

Each player/host has random tokens preventing session hijacking

### Automatic State Synchronization

The host broadcasts state updates on every state transition:

- Host serializes XState machine snapshot ( `getPersistedSnapshot()` )
- For each player, creates filtered view showing only information they should see
- Encrypts full state and includes in payload for recovery purposes
- Sends via WebSocket to all connected players
- If WebSocket send fails, state remains in host memory for retry on reconnection

## Consistency Model & State Management

### Authoritative State Pattern

The system uses **single-writer, multiple-reader** consistency:

- **Host** is the single source of truth (SSOT) for game state in a given room
- **Players** receive read-only projections of state relevant to them
- **All state mutations** flow through host's XState machine (deterministic transitions)
- **Backend** never modifies or interprets game state

This provides **strong consistency** from the host's perspective - all state transitions are serialized through a deterministic state machine, preventing race conditions.

### Event Sourcing via XState

The host uses XState (finite state machine library) to manage game state:

- Game logic encoded as declarative state transitions and guards
- Every player action becomes an event sent to the state machine
- State transitions are deterministic and testable
- Machine state is fully serializable for checkpointing

**Benefits for distributed systems**:

- Reproducible behavior (same events → same state)
- Easy to reason about distributed state
- Natural checkpoint boundaries (state snapshots)
- Guards prevent invalid state transitions even with network delays

### Eventual Consistency for Player Views

Players operate under **eventual consistency**:

- Player views lag behind host state by network latency + broadcast time
- Players accept that their view may be slightly stale
- Players never see inconsistent state (atomic state updates from host)
- For player actions (voting, submitting lies), optimistic UI updates with server reconciliation

## Security in a Distributed Context

### Encrypted State Distribution

To enable host recovery without allowing players to cheat:

1. Host encrypts full state using AES-256-GCM with key derived from host token
2. Encrypted state embedded in every state broadcast to players
3. Players store encrypted state but cannot decrypt it (don't have host token)
4. Only the original host can decrypt its own checkpoints

### Information Hiding via State Projections

Host sends different state views to different players:

- Players never see other players' articles during research phase
- During voting, players don't see who voted for what until reveal
- Expert identity hidden until submissions complete via delayed reveal timer
- Each player receives only the minimal information needed for their UI

## Connection to Distributed Systems Principles (I really hope this counts)

This project demonstrates several fundamental concepts from distributed systems and large-scale system design:

**Consensus & Coordination**: Single-master replication pattern (host as master, players as replicas)

**Fault Tolerance**:

- Graceful degradation on network partitions
- Checkpoint/restore pattern for crash recovery
- State machine allows for idempotent operations

**Data Distribution**:

- State sharding (each player receives only their view)
- Replication for disaster recovery (encrypted state on all clients)

**System Design Patterns**:

- Pub/Sub messaging (host publishes, players subscribe)
- Event-driven architecture (state machine reacts to events)
- Stateless service layer (backend can be easily replicated)

**Performance & Scalability**:

- Horizontal scaling via room isolation
- Client-side computation reduces server load

## Future Enhancements

1. Get rid of the host: Have a "leader" elected among the players that is responsible for state.
2. Redis-backed backend: Replace in-memory storage with Redis for multi-backend scaling
3. State compression: Implement LZ4/Snappy compression for state snapshots
4. Differential updates: Send only state deltas instead of full snapshots to reduce bandwidth
5. WebRTC peer-to-peer: Direct player-to-player communication without a full backend