Updated Code and Backend Implementation

Problems in My Current Code:
There are a few issues the code has with state management and logic:

- **Scattered State Logic:** Using multiple useState hooks for visitors, messages, and selectedVisitorId leads to a decentralized state management approach, and it is difficult to track and maintain the entire application state.
- **Repeated and Complex Logic:** State updates, especially those of visitors and messages, typically are repeated and interleaved logic executed within different useEffect hooks. This can cause issues, decreased readability, and greater complexity in understanding the code.
- **Unpredictable Simulation Timers:** The absence of a centralized state management solution, like an external state library or useReducer, prohibits scalability and maintainability for future feature development or more complicated data.

Improve the code:
In order to solve the problems mentioned earlier, it is strongly advised to utilize a useReducer hook to merge and make state management easier. The method has several advantages:

- **Centralized State:** One useReducer hook can handle all the relevant states like visitors, messages, and selectedVisitorId and offer one source of truth for the app's data.
- **Explicit State Transitions:** With explicit action types for different updates (e.g., adding a message, updating visitor status), useReducer maintains the state transitions explicit and more understandable.
- **Encapsulated Logic:** Complex state update logic can be moved to pure reducer functions, which makes it more readable, testable, and maintainable.
- **Performance Optimization:** With memoization techniques (e.g., useMemo), repeated operations like filtering and mapping can be optimized, and performance as a whole can be improved.
- **Better Scalability and Structure:** The useReducer pattern promotes a more modular structure, which is simpler to scale with extra features and intricate state interactions. Additionally, refactoring reusable logic out into custom hooks or utility functions can also enhance code structure.

Centralized State Management

- **Single Source :** Having all of the concerned state contained in a single useReducer hook provides you with one predictable app state object.
- **Example from the [React documentation]:**
- ***"Reducers enable you to place all of the state update logic for the connected slices of state in one place and make the logic easier to read and test."***

Explicit State Transitions

- **Clear Action Types:** Defining action types (e.g., ADD_MESSAGE, UPDATE_VISITOR) makes state transitions explicit and easier to trace.
- **Helpful line from the article:**
- ***"Reducers clarify state transitions, so you can clearly see how state changes as a result of actions."***

Performance Optimization

- **Memoization:** Using useMemo on expensive operations (like message filtering) prevents them from running when they are not needed, improving performance
- **From the article:**
- ***"Memoization can be used to avoid unnecessary recalculation when inputs have not been changed."***

Better Scalability and Organization

- **Scalable Architecture:** The useReducer pattern is scalable when introducing new features and handling complex state interactions.
- **Reusable Logic:** Breaking logic into utility functions or custom hooks also promotes code reusability and organization.

Aspect	PreviousCode Approach	New Code Approach
State Management	Multiple useState hooks	Single useReducer with explicit actions
State Updates	Scattered and repeated logic	Centralized, pure reducer functions
Real-time Updates	Simulated with timers	Real-time via WebSocket connection
Message Sending	Local state only	Optimistic UI + backend sync via WebSocket
Performance	Filtering in render	Memoized filtered messages
Scalability	Harder to extend	Easier to maintain and extend

Video Drive link - Screen Recording 2025-07-04 at 4.59.06 AM.mov

Github link - GitHub - vinay2003/c-zentrix-chatApp