# Design – coPlay Distributed System

This document outlines two architectural designs for the coPlay distributed system, addressing two distinct deployment scenarios. The first design targets a LAN-based organisation with 100 users using ZMQ for peer-to-peer communication. The second design targets a globally distributed deployment with 1 billion users, using Zookeeper to coordinate distributed webapp clients.

## 1. Design for Organisation Deployment (ZMQ Peer-to-Peer)

This design uses Python Flask web applications communicating over ZeroMQ (ZMQ) sockets in a peer-to-peer topology. Each instance runs independently, binds to a PULL socket, and connects to all others via PUSH sockets.

### Key Features:

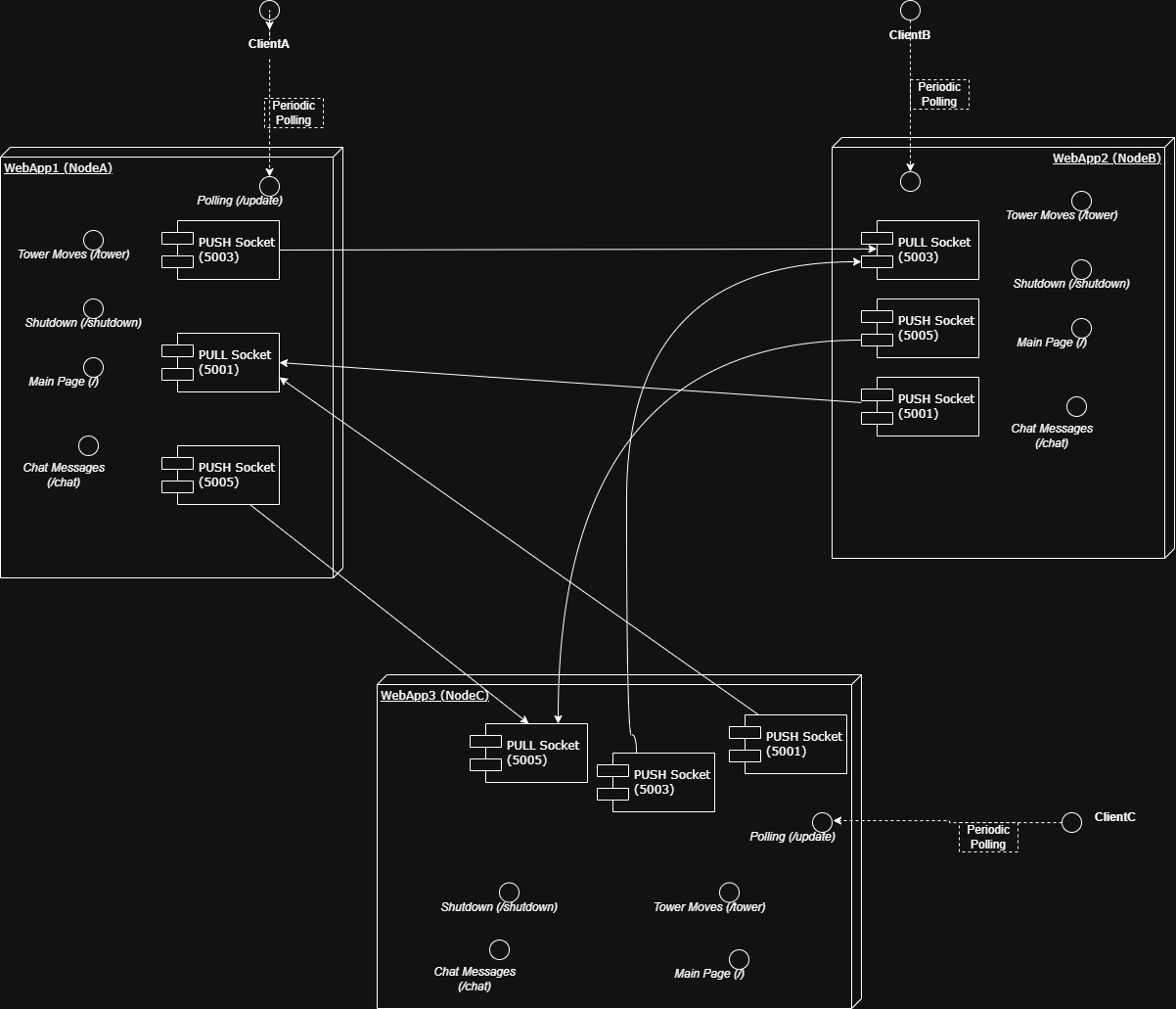
* Each webapp binds to a unique ZMQ PULL socket and connects to others using PUSH sockets.
* Broadcast messages using a rotating socket list to evenly distribute traffic.
* Polling mechanism from HTML (`GET /update`) ensures browser synchronization.
* Supports consistent game state and chat message delivery across multiple peers.
* Includes optional delay injection to simulate fault tolerance conditions.

### Limitations:

* Does not scale well beyond small group size due to manual port configuration.
* No global coordination or persistent state in event of crashes.
* Message loss or duplication possible due to non-reliable delivery semantics.

### Component Diagram:

The coPlay distributed system's Organization Deployment architecture is illustrated in the component diagram below.



Each WebApp node (Node A, Node B, and Node C) is a distinct Flask server that communicates over ZeroMQ (ZMQ) in this peer-to-peer communication model architecture. Every node has the following:

* PULL Sockets: For receiving messages from other nodes.
* PUSH Sockets: For sending messages to the other two nodes.
* Endpoints: To handle client requests, including:
* / (Main Page)
* /message (Chat Messages)
* /tower (Tower Moves)
* /update (Polling for Updates)
* /shutdown (Optional, for testing)

**Data Flow:**

* Incoming Messages: Every node uses its PULL socket to receive messages.
* Outgoing Messages: Using PUSH sockets, each node communicates with the other two nodes.
* Client Polling: In order to obtain real-time changes, clients frequently request the /update endpoint.

This architecture supports 100 users and guarantees scalability and fault tolerance in a LAN-based environment.

## 2. Design for Internet-Scale Deployment (Zookeeper Coordinated)

This design introduces Apache Zookeeper as a coordination layer for webapps. Each app becomes a Zookeeper client and uses znodes to register, watch, and broadcast state changes. Zookeeper ensures fault-tolerant coordination at scale.

### Key Features:

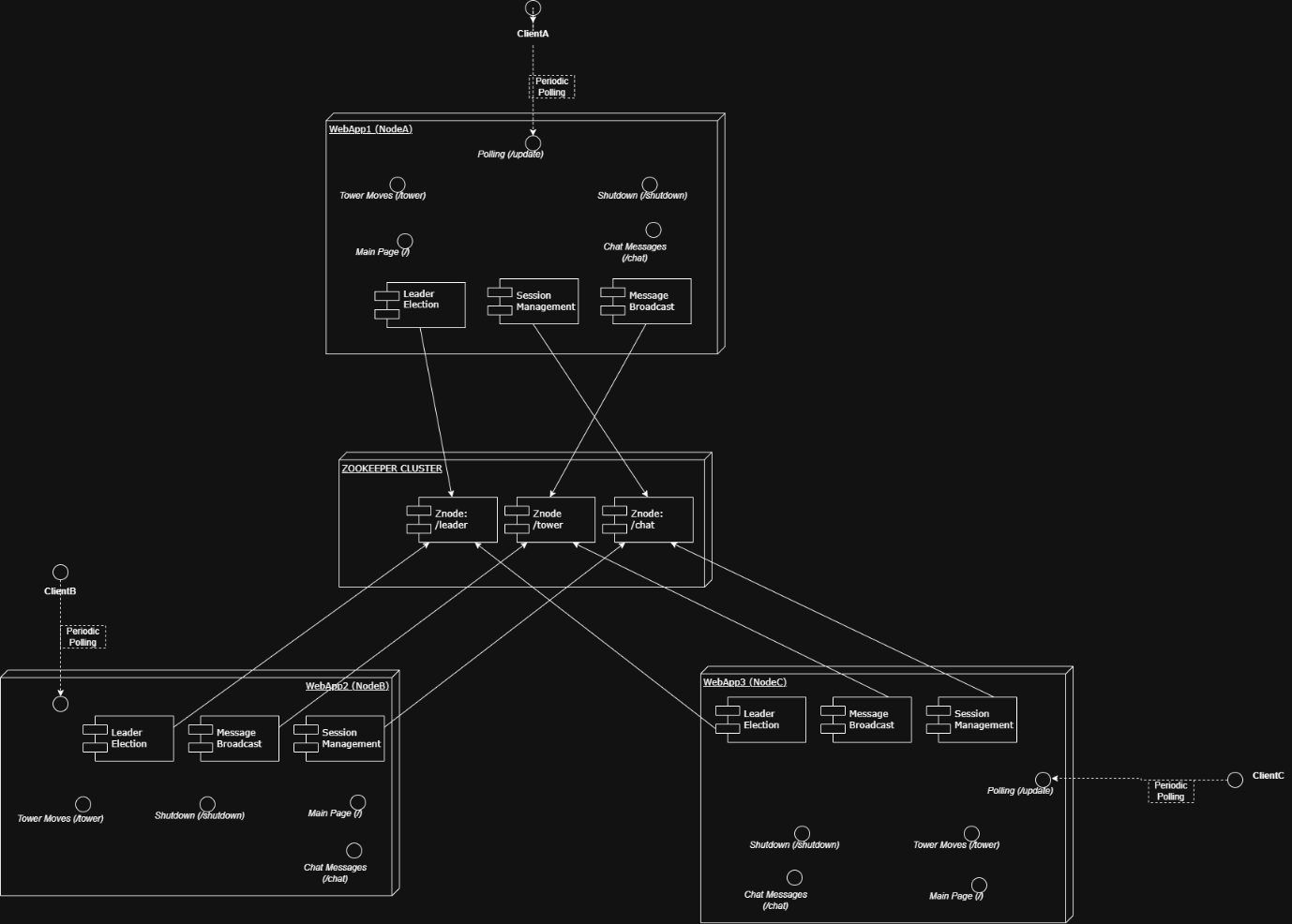
* Webapps register with Zookeeper and use ephemeral znodes to track session status.
* Leader election allows one node to be responsible for broadcasting updates.
* Clients watch znode paths for changes to reduce polling overhead.
* Tower moves and chat messages are posted as znode updates (JSON blobs).
* Zookeeper provides crash recovery and automatic node status detection.

### Limitations:

* Higher architectural complexity which requires a running Zookeeper cluster.
* Single-point of failure unless Zookeeper ensemble is replicated.
* Slight delay in reflecting state due to znode creation latency.

### High-Level Architecture Diagram:

This architecture allows the system to extend beyond local networks and manage worldwide deployments by integrating Zookeeper as a central coordinating layer. Each WebApp node (Node A, Node B, and Node C) connects to the Zookeeper Cluster for leader election and state synchronization.



Important Elements:

* **Zookeeper Cluster:** The main core for managing znodes for message broadcasting, leader election, and session management.
* **WebApp Nodes (Nodes A, B, and C)** are separate Flask web apps that link to the Zookeeper cluster in order to coordinate in real time.
* **Znodes:** Key information sources for monitoring and disseminating state changes
* **/chat**: Used to distribute chat messages and manage sessions.
* **/tower:** For game state updates and tower movement tracking.
* **/leader:** Used to elect a leader, enabling one node to assume command of important tasks.

Data Flow:

* Leader Election: By registering with the /leader znode, each node tries to take the lead. While the others keep an eye out for changes in leadership, the first node to successfully register takes over as the leader.
* Session Management: To guarantee uniform chat message delivery across all nodes, user sessions are monitored by the /chat znode.
* Message Broadcasting: To guarantee coordinated gameplay, game moves and state changes are transmitted via the /tower znode.
* Client Watches: To replicate the Zookeeper watch mechanism, external clients periodically request the /update endpoint for real-time updates. By doing this, extra overhead is avoided and each client is guaranteed a constant game and chat state.

The system can support millions of users with low latency because to this design, which enables it to scale beyond local networks. Additionally, it offers automated leader failover, which selects a new leader in the event that the current one fails.

Zookeeper's watch mechanism provides fault tolerance and high availability by ensuring that nodes can respond to changes in real-time.

## 3. Summary

The ZMQ design provides a lightweight peer-to-peer prototype suitable for controlled environments. The Zookeeper design introduces reliable coordination for real-world, global deployments. These designs balance simplicity, scalability, and resilience to meet the assignment requirements. The diagrams in this document are created with instructions from ChatGPT(<https://chatgpt.com/share/6821e92b-f72c-800c-b927-847c57724f00>)