# 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.

Diagram Instruction: Include a use case diagram here showing the actor 'User' with use cases like 'Send Chat', 'Click Tower', 'Poll Updates', and 'Reset Game'.

Diagram Instruction: Create a component diagram showing Flask webapps connected via ZMQ PUSH/PULL sockets, with HTML clients polling `/update`.

## 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.

Diagram Instruction: Include a class diagram modeling entities like `Webapp`, `ZookeeperClient`, `ZNode`, with attributes/methods like `broadcast()`, `watch()`, etc.

Diagram Instruction: Include an architecture diagram showing webapps connected to a Zookeeper ensemble with znodes like `/chat`, `/tower`, `/leader`.

## 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>)