Distri-place - Final report

Viljami Ranta, Ilari Heikkinen, Antti Ollikkala, Joni Pesonen

Table of Contents

Chapter 1

Introduction

This is the final report of our project Distri-place. Distri-place is a shared canvas app that allows users to distributedly work on the same graphical canvas to create works of art.

In this report we will discuss the project design and implementation goals and technology from the lens of implementing a working distributed system. We go into the core functionality and give a detailed review of how key distributed system elements such as leader election, consistency and fault tolerance are implemented, what are the roles of different nodes in our system and much more.

Chapter 2

Project Goals and Core Functionality

2.1 Project Objectives

The project objective was to be a shared canvas that works seamlessly with multitude of users each working on the canvas in real time. The distributed aspects of the project that we mainly were focusing on were replication among multiple nodes, global synchronization and availability, so that the users are always able to connect to our system.

2.2 Core Features

Distributed systems elements are mostly implemented via RAFT-algorithm. We concluded after our initial design (note in appendix) that with RAFT the project gets the key features that we expect from the distributed system. Those were the consistency and global synchronization, consensus via leader election and fault tolerance in case of errors happening that might compromise the current leader.

The core feature of our application for users is the ability to collaborate in coloring individual pixels of the canvas at the same time and seeing the canvas update in near real-time. Currently there is no limit to how often a user can color a pixel but in the original implementation (reddit.com/place) there are certain restrictions in place (time limit that user has to wait before being able to color again) to improve the user experience. Also for demo the size of our canvas is relatively small but in production environment our implementation could be scaled with small modifications to code to accommodate a much bigger canvas.

2.3 Potential Applications / Services Built on This Project

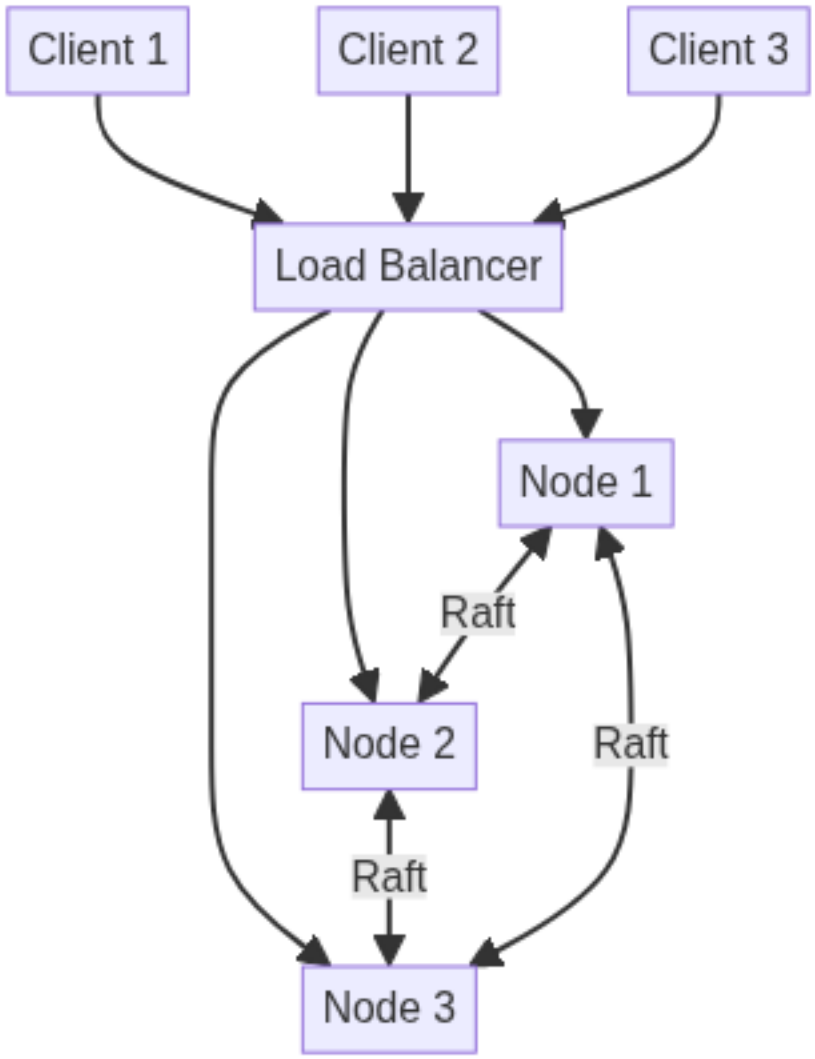
- As our application implements RAFT in a way that allows for strong consistency and availability and replication among nodes, there are a plethora of different possible uses for different kinds of applications. Our project may benefit for example from multi-raft scaling (more on that on scalability evaluation at #5). Our application has a very specific purpose as shared canvas so there are not a direct follow-up that could build onto this. However the underlying raft-implementation can be used for example in distributed databases, key-value stores or file management applications. So our project is mainly a standalone application.

Chapter 3

Design Principles

3.1 Architecture

distri-place uses a leader-based replicated architecture with 3 server nodes. Each node runs identical code but assumes different roles (leader, follower or candidate) based on Raft consensus.



- Client: Static HTML/JS page served locally with Nginx. Connects to load balancer for HTTP requests and WebSocket.
- Load Balancer: Simple Python round-robin proxy. Distributes client requests across the 3 nodes. In the demo running in melkki.
- Nodes: Python FastAPI servers. In the demo running in svm-11-1, svm-11-2, svm-11-3. Each node runs:
  - HTTP API (GET /client/pixels, POST /client/pixel)
  - WebSocket server for real-time updates
  - gRPC server for Raft communication with peers
  - In-memory canvas state (64x64 grid)

3.1.1 Why this architecture

- Full replication ensures any node can serve reads
- Leader-based writes provide strong consistency
- Raft unifies leader election and state replication in one protocol

3.2 Process

3.2.1 Startup Sequence

1. Node loads configuration (node ID, peer addresses, HTTP and GRPC ports) from environment variables
2. Canvas state-machine initialized as 64x64 in-memory grid
3. gRPC server starts for internal Raft communication
4. FastAPI HTTP server starts for client connections
5. Node initializes as Raft FOLLOWER with randomized election timeout (2.0-4.0s)
6. Raft event loop begins. If no heartbeat received within timeout, transitions to CANDIDATE

3.2.2 Leader Election Flow

1. Followers election timer expires -> becomes CANDIDATE, increments term
2. Votes for itself, broadcast `RequestVote`
3. If majority votes received -> becomes LEADER
4. Leader sends periodic `AppendEntries` heartbeats (every 1s) to maintain authority
5. If higher term discovered -> steps down to FOLLOWER

3.2.3 Pixel Placement Flow

1. Client sends POST `/client/pixel` {x, y, color, user_id} -> load balancer
2. Load balancer forwards request to any healthy node (round-robin)
3. Node processing:
   - follower -> forwards to leader via gRPC `SubmitPixel`, returns leader response
   - leader -> continues to step 4
4. Leader creates LogEntry{term, index, x, y, color} -> appends to local Raft log
5. Leader sends `AppendEntries` RPCs to all followers with new entry
6. Each follower:
   - validates log consistency (prev_log_index, prev_log_term)
   - appends entry to local log
   - responds success=true to leader
7. Leader tries to advance commit index:
   - counts replications: leader(1) + successful follower responses
   - majority achieved (2 or 3) -> advance commit index
8. Leader calls commit apply:
   - updates canvas state: `canvas.update(x, y, color)`
   - resolves pending commit future -> returns success=true to client
   - canvas triggers `on_update` callback -> broadcasts to WebSocket clients
9. On next `AppendEntries` (heartbeat), followers receive updated `leader_commit`:
   - update local `commit_index = min(leader_commit, last_log_index)`
   - apply the commits -> update local canvas state
   - broadcast pixel update to connected WebSocket clients

3.3 Communication

The system was two communication layers: client-server for client and server-server for Raft.

3.3.1 Client - Server

Protocol	Endpoint	Purpose
HTTP GET	`/client/pixels`	Fetch current canvas pixels
HTTP POST	`/client/pixel`	Submit a new pixel

Protocol	Endpoint	Purpose
WebSocket	`/ws`	Receive real-time pixel updates

When a client first loads the page, it fetches the full canvas from the LB via GET. When a user places a pixel, the client sends a POST to the LB.

The WebSocket connection is established on page load. When any pixel is committed on any node, that node broadcasts the update to all connected WebSocket clients. This allows users to see other changes in real-time.

The load balancer proxies both HTTP and WebSocket traffic. For WebSocket connections, it makes a connection to one of the backend nodes and forwards messages bidirectionally.

3.3.2 Server - Server

RPC	Purpose
`AppendEntries`	Log replication and heartbeat from leader to followers
`RequestVote`	Leader election voting during candidate phase
`SubmitPixel`	Forwarding pixel requests from follower to leader

GRPC is asynchronous, allowing nodes to handle multiple requests without blocking.

The RPC message definitiosn are availble in `server/proto/messages.proto`.

3.4 Mapping of Design to Source Code

3.4.1 Server

- `server/app/main.py` — Application startup and server initialization
- `server/app/app.py` — FastAPI application factory and configuration
- `server/app/config.py` — Environment configuration and settings
- `server/app/api/client/routes.py` — HTTP REST endpoints (GET /pixels, POST /pixel)
- `server/app/api/ws/routes.py` — WebSocket handlers for real-time updates
- `server/app/raft/node.py` — Core Raft algorithm implementation
- `server/app/raft/log.py` — Raft log data structure and operations
- `server/app/grpc/server.py` — gRPC server for interal communication
- `server/app/grpc/client.py` — gRPC client for internal communication
- `server/app/canvas/state.py` — Canvas state management (64x64 pixel grid)
- `server/app/client/manager.py` — WebSocket client connection management

3.4.2 Loadbalancer

- `loadbalancer/app/main.py` — Load balancer entry point
- `loadbalancer/app/handlers/http.py` — HTTP request proxy implementation
- `loadbalancer/app/handlers/websocket.py` — WebSocket proxy for real-time communication
- `loadbalancer/app/balancer/strategy.py` — Round-robin load balancing algorithm

3.4.3 Client

- `client/index.html` — Static HTML frontend with canvas UI
- `client/app.js` — JavaScript client for canvas rendering and API calls

Chapter 4

System Functionalities

4.1 Global State Management

The global state of the system consists primarily of the shared canvas, represented as a fixed-size 64×64 grid of pixels. Each pixel is defined by its coordinates and color value. This state is fully replicated across all nodes in the Raft cluster.

Rather than directly modifying the canvas state, all changes are expressed as log entries in the Raft log. A pixel placement operation is encoded as a deterministic command (x-coordinate, y-coordinate, color), which is appended to the leader's log. Once the log entry is committed, it is applied to the in-memory canvas state on all nodes in the same order.

This approach ensures that:

- All nodes eventually reach the same canvas state
- State transitions are deterministic and replayable
- New or recovering nodes can reconstruct the full canvas state by replaying the log

Reads (fetching the current canvas) can be served by any node, since all nodes maintain a replicated and up-to-date view of the global state once entries are committed.

4.2 Consistency and Synchronization

Distri-place provides strong consistency for write operations. At any given time, there is exactly one leader node responsible for accepting and ordering state changes. All pixel placement requests are either handled directly by the leader or forwarded to it if they arrive at a follower.

Consistency is achieved through the Raft log replication mechanism:

1. The leader appends a new pixel operation to its log.
2. The leader replicates the log entry to followers using AppendEntries RPCs.
3. Once a majority of nodes acknowledge the entry, it is considered committed.
4. The committed entry is then applied to the canvas state on all nodes.

Synchronization between clients is handled via WebSockets. After a log entry is committed and applied, the leader broadcasts the pixel update to all connected clients. This ensures that all users see updates in near real-time and in the same order, preserving a consistent view of the canvas.

4.3 Consensus Mechanism

Consensus in Distri-place is implemented using the Raft consensus algorithm. Raft is responsible for both leader election and agreement on the order of state changes.

4.4 Fault Tolerance and Recovery

Fault tolerance is achieved through replication and automated leader re-election. The system can tolerate failures of up to FLOOR((N-1)/2) nodes, where N is the total number of nodes in the cluster (in our case, one node out of three).

4.4.1 If the leader fails:

- Followers stop receiving heartbeats.
- After a timeout, a new leader election is triggered.
- A new leader is elected without client intervention.

4.4.2 If a follower fails:

- The leader detects failure via failed AppendEntries RPC.
- Commits continue as long as majority quorum is maintained.
- The system remains fully operational for reads and writes.

4.4.3 Limitations

When a crashed node restarts, it rejoins the cluster with an empty state (no pixels). Our app does not persist the Raft log or state-machine to disk, so a recovering node starts with no knowledge of previous operations.

Recovery works through Raft's normal log replication mechanism:

1. The recovering node starts as FOLLOWER with an empty log
2. The leader sends AppendEntries RPCs and detects log inconsistency
3. The leader decrements `next_index` until logs match (index 0 for empty node)
4. All log entries are then replayed to the recovering node
5. As entries commit, the node rebuilds its canvas state by applying each pixel operation

This means that recovery time can take very long depending on the amount of pixels. In the current implementation, during this time the node will show an empty canvas until it catches up.

In production we would improve this with persistent log storage and snapshots for sync.

Chapter 5

Scalability Evaluation

Our system is technically highly scalable. However scaling with just one leader is not ideal for the use cases of our project and our project goals. More of this just down below at #6

Instead a better approach to scaling for our project would be for example to implement a multi-raft solution where the canvas is partitioned into multiple areas that each implement their own raft-environment. Each area would have their own leader and and followers for replication and fault tolerance. This approach would be a good option if our userbase were to grow very large and a single raft-cluster would experience congestion because of that. So the approach to scaling would have to be considered case by case and scaling just the single raft-cluster could be a good idea to a particular level.

5.1 Adding More Nodes

Adding a new node to the cluster is straightforward. A new node can be started with `make start-node` in the server directory, where the environment variables specify the node ID, ports, and peers. As long as the new node is configured with the correct peer hosts and all existing nodes are updated to include the new peer, the node will join the cluster and begin participating in Raft consensus.

When a new node joins, it starts as a follower and the current leader begins sending AppendEntries RPCs to bring it up to date. The node catches up by receiving and applying all log entries.

Adding nodes affects the quorum requirement for commits. Raft requires a majority of nodes to agree before committing an entry. With 3 nodes, quorum is 2. With 5 nodes, quorum increases to 3. This means more nodes improve fault tolerance (a 5-node cluster can survive 2 failures instead of 1) but also slightly increase commit latency since the leader must wait for more acknowledgments.

5.2 Bottlenecks

- Leader bandwidth: All writes must pass through a single leader
- Full replication: Each node stores the complete 64×64 canvas
- Commit overhead: Every pixel requires majority acknowledgment
- WebSocket connections: Each node maintains connections to all its clients

5.3 Possible improvements

- Divide the canvas into regions each managed by an independent Raft cluster.
- Batching: Group multiple pixel operations into a single log entry.
- Snapshotting: Periodically snapshot canvas state to speed up recovery.

Chapter 6

Performance Evaluation

6.1 Latency

Pixel placement latency can be used to quantify the general perfomance of the system. It measures the time from client clicking a pixel to all clients seeing the updated canvas. The latency is built from a sequence of events:

Client click -> Load balancer -> Server receives -> Raft commit -> Canvas update -> WebSocket broadcast -> Client render

Phase	Description	Duration
Client to load balancer	HTTP request from client to melkki	10-30ms
Load balancer to node	Round-robin forwarding to a server node	10-30ms
Leader forwarding	If request hits follower, forwarded to leader via gRPC	0-50ms
Raft commit	Leader replicates to majority and commits	100-1000ms
WebSocket broadcast	Server pushes update to all connected clients	0-30ms

Total latency is typically 100-1100ms depending on network and if the request hits the leader or a follower.

The most significant portion of latency comes from the Raft commit phase, which requires at minimum one round-trip to a majority of nodes.

The best improvement would be batching pixels into a single log entry.

However, the latency is not visible to user since the client UI updates immediately on click, optimistically assuming success. If the update fails, the client will revert the update.

6.2 Throughput

Thoughput is the other important measure to quantify perfomance of the system. Our implementations uses a single leader for all writes for the full canvas. This is an obvious bottleneck when you add more clients. Successful commits per second measures the throughput for the system. Improving throughput could be done by sharding the canvas for multiple leaders. For example dividing the canvas into four regions which are managed by their own leaders.

6.3 Setup

Our observations are based on manual testing with the following setup:

- 3 server nodes running on svm-11.cs.helsinki.fi, svm-11-2.cs.helsinki.fi, and svm-11-3.cs.helsinki.fi
- Load balancer running on melkki.cs.helsinki.fi
- Client accessed via localhost through SSH tunnel port forwarding to melkki
- Latency measured using browser developer tools network tab

6.4 Improvements

Implemented:

- gRPC with keepalive settings for persistent connections
- Async Python with FastAPI for non-blocking I/O, allowing concurrent request handling
- In-memory canvas state for fast reads without disk access
- WebSocket connection proxying through load balancer for real-time updates

Future:

- Batching: Group multiple pixel operations into a single log entry.
- Snapshotting: Periodically snapshot canvas state to speed up recovery.

Chapter 7

Key Enablers and Lessons Learned

One of the key enablers of the project was the early decision to base the system on a well-established consensus algorithm. Choosing Raft significantly simplified the design of leader election, replication, and failure handling. Instead of implementing separate mechanisms for consensus, synchronization, and fault tolerance, Raft provided a good framework that covered all the concerns in a well documented way.

Another important enabler was the clear separation of concerns in the system architecture. Client-facing logic, consensus logic, and state management were implemented as distinct components. The modularity made the system easier to reason about, debug, and extend. It also allowed different group members to work on separate parts of the system in parallel without excessive coupling.

From an implementation perspective, using FastAPI and gRPC proved effective. FastAPI as a modern framework enabled rapid development of a clean client API, while gRPC provided an efficient and type safe RPC communication layer for inter node communication.

A key lesson learned during the project was related to performance expectations in distributed systems. Initially, it was assumed that distributing the system across multiple nodes would naturally improve performance. However, it became clear that Raft is not optimal for workloads where distribution is expected to increase performance. In a leader-based consensus system, adding more nodes can actually might decrease performance due to increased coordination overhead, additional network communication, and the fact that all write operations must still pass through a single leader. While Raft scales well in terms of availability and reliability, it does not inherently scale write throughput. This realization motivated the discussion of alternative designs, such as multi-Raft or sharded architectures, where load can be distributed across multiple leaders.

Chapter 8

Groupwork

- Everyone participated equally to the project planning phase and drawing the first sketches of our project. The big decisions were made together.
- Antti should get extra mention for the bigger effort on raft-implementation and Viljami for setting up the environment for the demo
- Overall we are happy with how everyone participated to the project

Chapter 9

Appendices

9.1 Changes to our initial plan

- In our original project design-plan we considered using bully algorithm for the consensus. Instead we decided to implement the RAFT algorithm as it included the core functionalities that we wanted from our distributed system.
- Initially we thought about using TypeScript and React in the Front-end and Flask in the Back-end. However we decided to create the front-end without dedicated framework and in the back-end we used FastAPI with Python and added gRPC for communication between nodes.