report p2

Mounir Samite 2026

Table of Contents

1 Problem analysis

The objective of this project is to transform a centralized, single user pixel art application into a distributed, collaborative system where multiple users can simultaneously view and modify a shared pixel grid. The baseline application provides the basic architecture, including a PixelGrid data structure, event driven user interactions through listener interfaces (pattern Observer), and a graphical interface with brush management.

1.1 State consistency and determinism

The fundamental challenge lies in maintaining a deterministic shared state across all distributed clients. In the centralized version, state mutations occur directly through the PixelGrid.set() method when a user clicks on a cell. In the distributed context, this approach is insufficient as concurrent modifications from multiple users could lead to inconsistent states. The solution must adopt an event driven architecture where state changes are triggered exclusively by messages and events processed locally, ensuring that each client's state transitions follow a predictable, reproducible sequence.

1.2 Message ordering

Preserving the causal ordering of events is critical for consistency guarantees. The specification mandates that if two events ev1 and ev2 occur such that ev1 $\rightarrow$ ev2 for one user, this ordering must be maintained for all users. Without proper message ordering mechanisms, scenarios such as a user changing their brush color before painting a pixel could result in other users observing the pixel painted with the wrong color. This requires implementing either logical clocks or leveraging ordering guarantees provided by the chosen middleware technology.

1.3 Dynamic peer discovery

The system must support dynamic p2p membership, allowing users to join by contacting any existing participant. The architecture must handle peer discovery, state synchronization for newly joined users, and graceful handling of user departures without disrupting ongoing collaboration.

1.4 Cursor awareness

Beyond grid modifications, the system must propagate cursor positions to provide awareness of where other users are pointing. The baseline application already includes `MouseMovedListener` and `BrushManager` components that track and visualize multiple brushes. These must be extended to broadcast cursor movements across the network while managing the performance implications of high frequency position updates.

2 MOM approach

The MOM implementation uses RabbitMQ as the message broker to enable distributed collaboration through a publish-subscribe pattern. This approach decouples clients from direct p2p communication, centralizing message routing through the broker while maintaining an eventdriven architecture where all state changes originate from received messages.

2.1 Architecture

2.1.1 Broadcast communication with fanout exchange

The system uses a RabbitMQ fanout exchange named pixel_art to broadcast all events to every connected client. Each client establishes a connection to the broker on localhost, declares the fanout exchange, and binds an exclusive queue to receive all broadcast messages. This architecture ensures that every action performed by one client is disseminated to all other participants without requiring explicit knowledge of peer addresses.

2.1.2 Client identity and message protocol

Each client generates a unique UUID upon initialization (CLIENT_ID) to distinguish its own messages from those of other participants. Messages follow a simple pipe format: type|senderId|x|y|color, where the type field indicates the event category. This lightweight protocol supports five core event types:

- **join**: Announces a new client's entry with initial brush position and color
- **leave**: Signals graceful disconnection via a shutdown hook
- **move**: Propagates cursor position updates for awareness visualization
- **draw**: Broadcasts pixel modifications with coordinates and color
- **colorChange**: Notifies peers of brush color updates

2.2 State management

2.2.1 State updates

The implementation strictly adheres to the principle of deterministic state changes through message processing. Local user interactions trigger both immediate local state updates and message broadcasts, ensuring the initiating client sees changes without network latency. When the `handleIncomingMessage` method processes incoming events, it filters out the client's own messages using the senderId comparison, preventing echo effects.

2.2.2 Grid serialization for late joiners

To synchronize newly joined clients, the system implements a state transfer mechanism using JSON serialization. When a client receives a join message, it serializes its current PixelGrid state using Jackson's ObjectMapper and broadcasts a grid_status message containing the complete grid array. This allows late joiners to reconstruct the shared canvas state rather than starting from an empty grid. The PixelGrid class provides `serializedGrid()` and `setGrid()` methods specifically for this purpose.

2.2.3 Brush management and Cursor awareness

The BrushManager was refactored from a listbased structure to a `Map<String, Brush>` to associate each remote cursor with its client ID. When move events arrive, the system checks whether a brush already exists for that sender and either updates its position or creates a new brush entry. This design gracefully handles scenarios where cursor movements arrive before join messages, dynamically registering unknown participants.

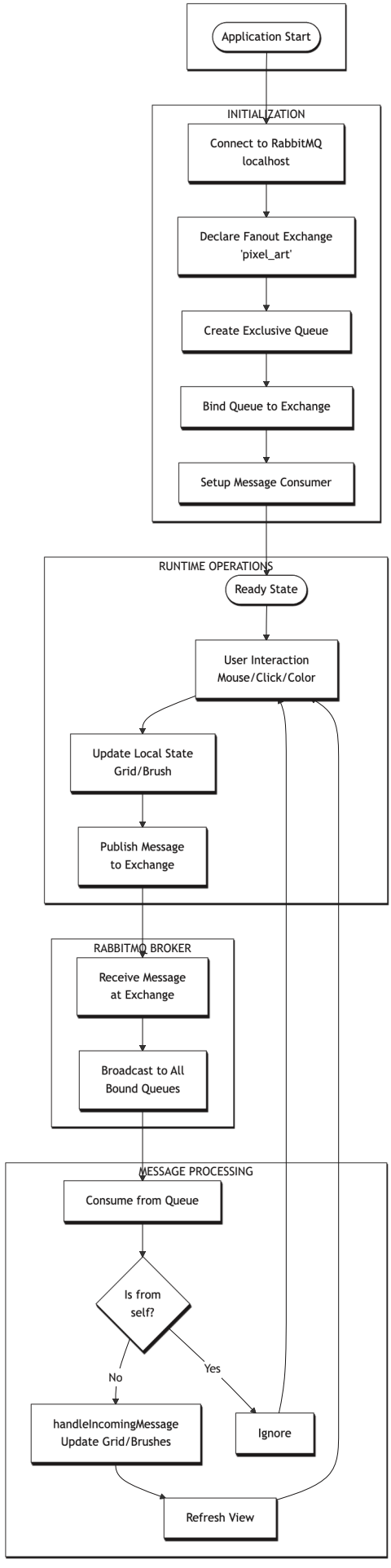
2.3 Consistency and ordering

2.3.1 Message ordering

RabbitMQ provides FIFO ordering guarantees for messages published to the same queue from a single connection. Since all clients publish to the fanout exchange and consume from their exclusive queues, messages from any single sender maintain causal order for all receivers. However, this implementation does not implement logical timestamps, meaning that concurrent events from different clients may be observed in different orders by different recipients, a tradeoff accepted in favor of simplicity.

2.3.2 Disaster recovery limitations

The current implementation lacks persistence mechanisms. If all clients disconnect, the shared pixel art state is lost. Additionally, there is no brokerside message persistence configured, so broker failures would result in message loss. The shutdown hook attempts graceful disconnection by sending leave messages, but unexpected client termination results in abandoned brush cursors remaining visible.


Application Start
INITIALIZATION
Connect to RabbitMQ localhost
Declare Fanout Exchange 'pixel_art'
Create Exclusive Queue
Bind Queue to Exchange
Setup Message Consumer
RUNTIME OPERATIONS
Ready State
User Interaction Mouse/Click/Color
Update Local State Grid/Brush
Publish Message to Exchange
RABBITMQ BROKER
Receive Message at Exchange
Broadcast to All Bound Queues
MESSAGE PROCESSING
Consume from Queue
Is from self?
No
Yes
handleIncomingMessage Update Grid/Brushes
Ignore
Refresh View

3 Java RMI approach

Unlike the MOM approach's brokercentric structure, this solution designates one peer as the coordination leader while maintaining p2p capabilities through RMI registry discovery.

3.1 Architecture

3.1.1 Leader

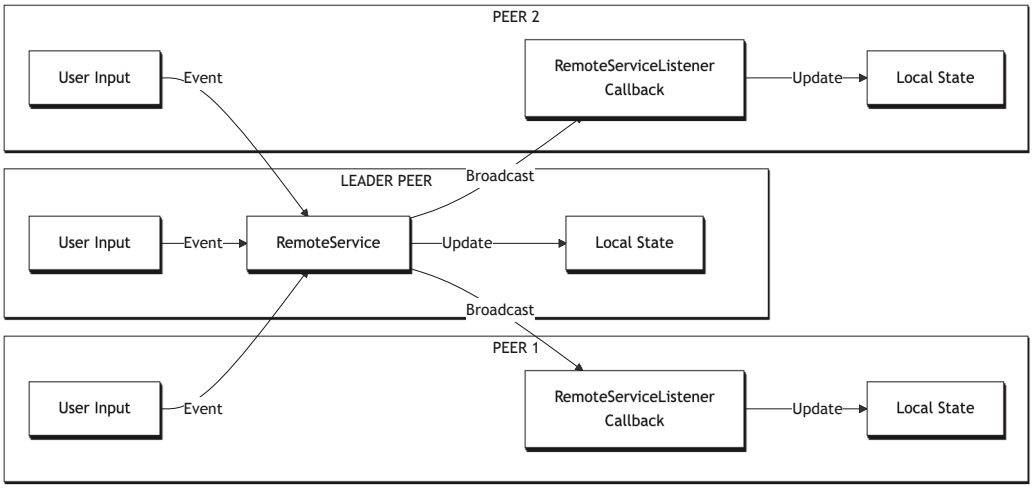
The system employs a centralized coordination model where the first client to start becomes the leader (peer ID 0) and instantiates a `RemoteServiceImpl` object bound to the RMI registry as *"rsObj"*. Subsequent clients attempt to lookup this remote service reference; if found, they join as followers by invoking the join() method, which registers their callback listener and assigns them a unique integer peer ID. This architecture centralizes event sequencing at the leader while distributing event processing and state visualization across all peers.

3.1.2 Remote callbacks

The design implements the Observer pattern through RMI callbacks. Each follower exports a `RemoteServiceListener` stub that the leader stores in a `ConcurrentHashMap<Integer, RemoteServiceListener>`. When the leader's `handleEvent()` method processes an event, it invokes methods like `notifyBrushMoved()`, `notifyPixelDrawn()`, or `notifyBrushColorChanged()` on each registered listener. The leader itself also implements a `RemoteEventHandler` to update its own local state through the `informLeader()` method, ensuring consistency between its coordinator role and participant role.

3.1.3 Event management

Events are encapsulated in `RemoteEvent` objects containing an `EventType` enumeration and a `BrushDTO` data transfer object. The `BrushDTO` class implements Serializable and carries peer identity, position coordinates, and color information, all immutable fields to prevent remote state corruption. The `EventType` enum defines five event categories matching the MOM implementation: ADD, MOVE, DRAW, COLOR_CHANGE, and LEAVE.



3.2 Leader election and Fault recovery

3.2.1 Leader election algorithm

For the leader election I tried to apply the chang-roberts algorithm. The election algorithm gets triggered when the leader peer ID appears in a LEAVE event. The leaderLeft() method searches backward from the highest assigned peer ID to find the next eligible leader still present in the listenersMap. Once identified, the current leader broadcasts a notifyNextLeader() callback containing the new leader ID and a snapshot of the listener map.

3.2.2 Leader transition handling

When followers receive the `onNextLeaderElection()` callback, each checks whether its own peer ID matches the designated leader ID. The chosen successor instantiates a new RemoteServiceImpl, rebinds it to the registry under both *"rsObj"* and *"rsObj"* + *leaderId*, and assumes coordination responsibilities. Nonelected peers spawn a background thread to repeatedly attempt reconnection to the new leader's registry entry until successful.

3.3 Consistency and ordering

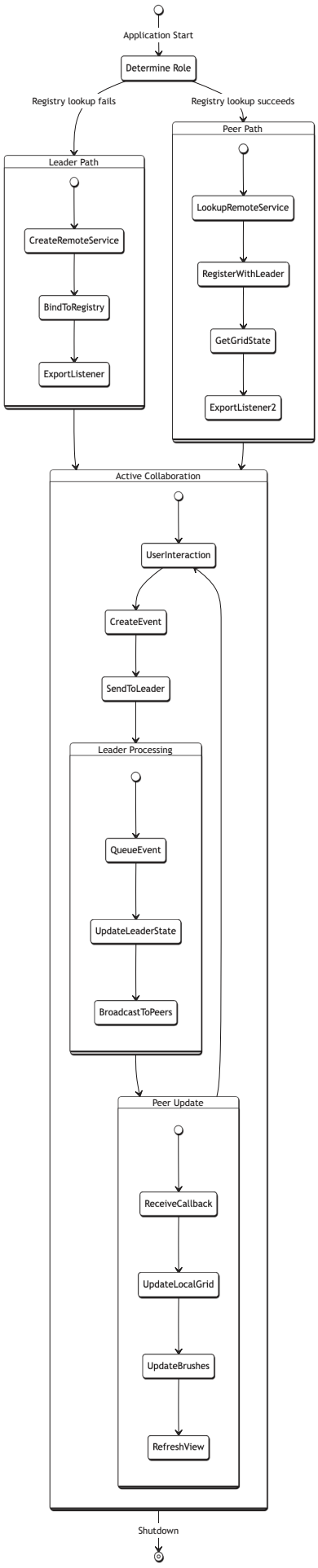
3.3.1 Total ordering through leader process queue

All events flow through the leader `processQueue()` method marked synchronized, establishing a total order. This guarantees that if one peer observes event A before event B, all

other peers will observe the same ordering. The tradeoff is that the leader becomes a single point of failure.

3.3.2 Remote exception handling

The implementation uses synchronized blocks on callback methods and brush updates to prevent interleaving. However, RemoteException instances during broadcasts are caught and logged without retries, accepting potential message loss in exchange for system availability. The leaderAvailable flag prevents clients from dispatching events during leader transitions.


Application Start
Determine Role
Registry lookup fails
Registry lookup succeeds
Leader Path
CreateRemoteService
BindToRegistry
ExportListener
Peer Path
LookupRemoteService
RegisterWithLeader
GetGridState
ExportListener2
Active Collaboration
UserInteraction
CreateEvent
SendToLeader
Leader Processing
QueueEvent
UpdateLeaderState
BroadcastToPeers
Peer Update
ReceiveCallback
UpdateLocalGrid
UpdateBrushes
RefreshView
Shutdown

4 CAP theorem comparison (a brief analysis)

The **MOM solution** is an AP system (availability + partition tolerance), prioritizing system responsiveness over strict consistency. Clients can publish events asynchronously even during network issues, but concurrent operations may be observed in different orders by different users, resulting in eventual consistency. The fanout exchange broadcasts messages without global coordination, allowing temporary state divergence that converges over time.

The **RMI solution** is a CP system (consistency + partition tolerance), enforcing strong consistency through the leader's event queue. All events are totally ordered, every client observes identical state transitions in the same sequence. However, the leader is a single point of failure: if it becomes unreachable, the entire system halts, sacrificing availability to maintain consistency.