Federated Skillshare

Final Report for the Distributed Systems Course – A.Y. 2025/2026

Yeray González Menéndez
yeray.gonzalez@studio.unibo.it

Jorge Vigil Bravo
jorge.vigilbravo@studio.unibo.it

July 7, 2026

Skillshare is a peer-to-peer platform in which users exchange knowledge by publishing skill announcements, requesting exchanges, communicating through chat, and reviewing one another. The original application used one server and one MapDB database. This project transforms it into a federated distributed system demonstrated through two local Docker instances, each with independent users and persistent state. Instances exchange domain events through HTTP using JSON representations and identify users as `user@instance`. Selected data concerning announcements, exchange requests, chat messages, and reviews is replicated asynchronously. Local operations remain available when a peer is unreachable, while a persistent outbox, manual and automatic retries, stable event identifiers, and a processed-event inbox ledger support eventual consistency and idempotent handling. Federated chat uses persistent Lamport logical clocks to obtain a causality-compatible deterministic ordering without depending on synchronized physical clocks. Reviews are propagated to the reviewed user's home instance and contribute to local reputation. The prototype therefore provides a concrete study of federation, replication, partial failure, recovery, logical time, and the trade-off between availability and immediate consistency.

Disclaimer

During the preparation of this report, the authors used OpenAI Codex to assist with repository analysis and the initial organization and drafting of the text. The authors reviewed and edited the resulting content as needed and take full responsibility for the content of the final report and artifact.

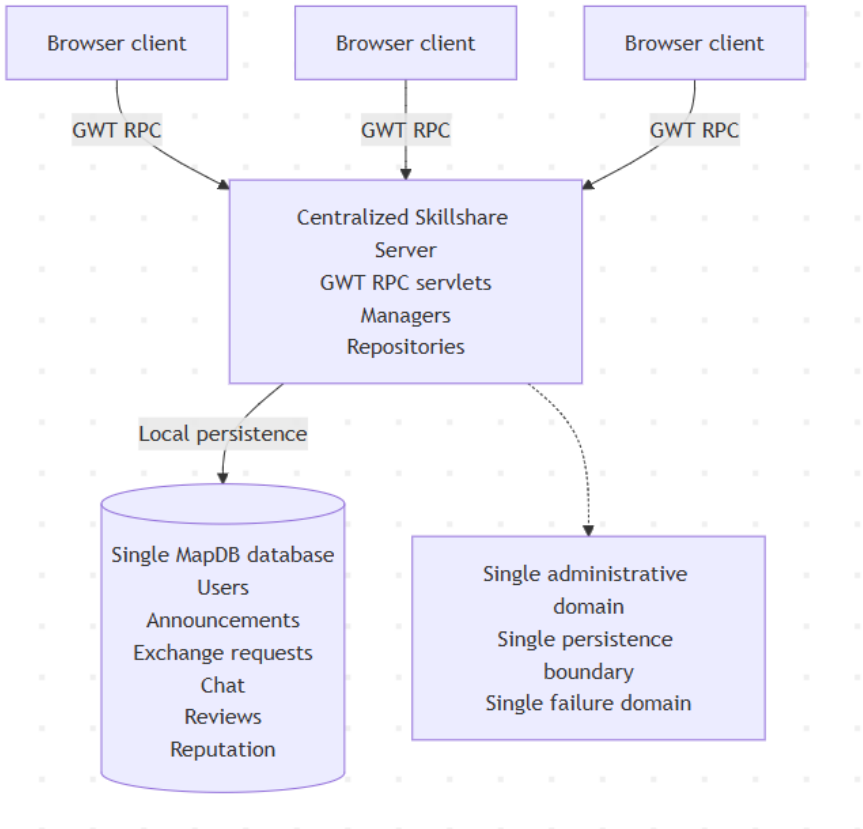

Figure 1: Original centralized Skillshare architecture. All users and domain data were managed by a single server and one MapDB database, creating a single administrative and failure domain.

1 Concept

Skillshare is a web application with a graphical browser interface. It supports reciprocal skill exchange: a participant can publish a skill they can teach, describe a skill they would like to learn, browse the marketplace, request an exchange, communicate with another participant after acceptance, and leave a review. The same person may act as both learner and teacher, so the product models an exchange community rather than a conventional e-learning store.

The original product was developed as a centralized web system. A GWT client communicated with one Java backend through asynchronous GWT RPC, and all users, announcements, requests, chat messages, reviews, and reputation data were stored in one MapDB database. This was suitable for the original functional scope, but it also created a single administrative and failure domain. If the server or database failed, the whole community became unavailable.

For the Distributed Systems course, Skillshare was evolved into a federation of au-

tonomous instances. The final demonstration scope consists of two local instances, `inst-a` and `inst-b`. They run the same application code but have different identities, users, databases, ports, and peer configuration. Cooperation happens through messages rather than shared memory or a shared database.

The users are placed at the edges of the system and interact with their own local instance through a browser. In the demonstration, one user logs in at `http://localhost:8080` and another at `http://localhost:8081`. Each instance owns its local users and persistent MapDB state. Federated operations allow users from one instance to see remote announcements, request exchanges, chat after acceptance, and review remote participants.

Distribution is needed to remove the assumption that all users belong to one server and one database. It introduces local autonomy, resource sharing across instances, partial failure tolerance, and asynchronous replication. A peer may be temporarily unreachable, but the local instance should continue serving its users. When connectivity returns, retained events should allow the remote view to converge. The project therefore focuses on federation, replication, failure and recovery, causality, and the trade-off between local availability and immediate consistency.

2 Requirements Elicitation and Analysis

The requirements describe what the distributed extension should provide. They are divided into functional, non-functional, and implementation requirements. Each requirement is paired with an acceptance criterion used later in validation.

Functional requirements

1. **FR1 – Independent instances.** Two Skillshare instances shall run independently with different instance identifiers and databases. Acceptance criterion: starting the Docker deployment creates two services with separate ports, identifiers, and data directories.

2. **FR2 – Federated identity.** A user involved in federation shall be identified by a local name and home instance, displayed as `user@instance`. Acceptance criterion: same-named users on different instances remain distinguishable in exchanged data and user interface labels.

3. **FR3 – Announcement federation.** Creating, updating, or deleting an announcement shall produce a corresponding event for known peers. Acceptance criterion: a created, updated, or deleted announcement on one instance is reflected on the configured peer after delivery.

4. **FR4 – Federated exchanges.** A local user shall be able to request a remote announcement, and acceptance or rejection shall propagate to the requester's instance. Acceptance criterion: a remote request appears on the owner's instance and the final status converges on both sides.

5. **FR5 – Federated chat.** Participants in an accepted cross-instance exchange shall be able to exchange replicated chat messages. Acceptance criterion: messages sent by each participant are visible in the conversation replica of the other participant.

6. **FR6 – Logical chat order.** Federated chat shall use logical timestamps to produce the same deterministic display order on both participant instances. Acceptance criterion: chat replicas sort messages using Lamport timestamps and deterministic tie-breakers.

7. **FR7 – Federated reviews.** A review of a remote participant shall reach that participant's home instance and contribute to reputation. Acceptance criterion: a remote review is stored at the reviewed user's home instance and appears in reputation computation.

8. **FR8 – Recovery.** Failed outgoing events shall remain persistent and shall be retryable both automatically and manually. Acceptance criterion: stopping a peer leaves failed outbox entries that can be delivered after the peer restarts.

9. **FR9 – Duplicate suppression.** A receiver shall avoid applying the same federation event more than once under ordinary retry. Acceptance criterion: repeated delivery of the same event ID is acknowledged without duplicating its effect.

Non-functional requirements

1. **NFR1 – Local availability.** A peer failure shall not roll back a valid local operation solely because federation delivery failed. Acceptance criterion: a valid local operation succeeds even when the peer is offline.

2. **NFR2 – Eventual consistency.** Retained events shall allow replicas to converge after temporary peer unavailability. Acceptance criterion: after retry succeeds, the peer receives the missing state changes.

3. **NFR3 – Autonomy.** Each instance shall own its users and local data without relying on a shared database or coordinator. Acceptance criterion: the system contains no shared MapDB file and no central coordinator.

4. **NFR4 – Durability.** Local domain state, the outbox, the processed-event ledger, and the Lamport clock shall use persistent MapDB storage. Acceptance criterion: those structures survive an application restart.

5. **NFR5 – Reproducibility.** The two-instance demonstration shall be launchable with Docker Compose. Acceptance criterion: the deployment starts from the project `app` directory with `docker compose up -d -build`.

6. **NFR6 – Compatibility.** Existing single-instance data with missing federation fields should continue to be treated as local data where practical. Acceptance criterion: legacy objects are handled as local data rather than crashing normal workflows.

Implementation requirements

1. **IR1 – Java and GWT compatibility.** The project shall preserve the existing Java/GWT architecture inherited from the Software Engineering project. This is justified by course continuity and by the need to extend the existing codebase rather than rewrite the product.

2. **IR2 – Persistent embedded storage.** The project shall continue using MapDB for persistent local state because it was already used by the application and allows each Docker instance to keep an independent local database file.

3. **IR3 – Containerized demonstration.** The project shall provide a Docker Compose deployment for two local instances to make evaluation reproducible on a clean machine.

Glossary

Instance An autonomous Skillshare server with its own identity, users, database, and peer configuration.

Federated identity A user identity composed of username and home instance, rendered as `user@instance`.

Federation event A JSON-serializable domain event sent by one instance to another through HTTP.

Outbox A persistent local store of outgoing federation events and their delivery status.

Inbox ledger A persistent local store of processed event IDs used to suppress duplicates.

2.1 Relevant Distributed System Features

The most relevant features are autonomy, availability under partial failure, asynchronous replication, dependability, and logical time. Complete transparency is not the main objective because the system deliberately exposes some distributed-system effects, such as temporary inconsistency and recovery after retry. Security is relevant because users and reviews are sensitive application data, but strong inter-instance trust remains a future work item.

The CAP theorem states that a distributed data system cannot simultaneously guarantee consistency and availability during a network partition [3]. Federated Skillshare chooses an availability-oriented behavior for the demonstrated operations: the local state is committed before delivery to peers is known to have succeeded. During peer unavailability, the remote view may be stale. After communication is restored, retry is intended to make replicas converge.

Feature	Project mechanism	Scope or limitation
Transparency	Users interact with their local web instance while remote data appears in the marketplace.	Failures and replication are not completely hidden: stale data can exist until retry succeeds.
Fault tolerance and dependability	Persistent outbox, retry scheduler, manual retry endpoint, and inbox ledger.	At-least-once delivery, not exactly-once delivery.
Scalability	Fixed peer list and event broadcast support a small federation.	Not intended for Internet-scale federation or large peer discovery.
Security and trust	Local login and domain authorization rules remain in the web application.	No cryptographic peer authentication or signed federation events.
Resource sharing	Instances share selected announcements, exchange requests, chat messages, and reviews.	Users, profiles, and authentication remain local.
Openness and interoperability	Federation uses HTTP and JSON, which are standard technologies.	It is not a full ActivityPub-compatible server.
Evolvability and maintainability	The federation layer is separated into event, client, inbox, repository, and manager components.	More versioning and capability negotiation would be needed in production.
Performance and communication efficiency	Asynchronous events avoid synchronous distributed transactions.	Broadcast and polling-style refresh are acceptable only for the small demonstration.
Economy and cost	Docker Compose reuses the same application image with different configuration.	There is no cloud deployment or production operations model.

Table 1: Distributed system features mapped to concrete project mechanisms.

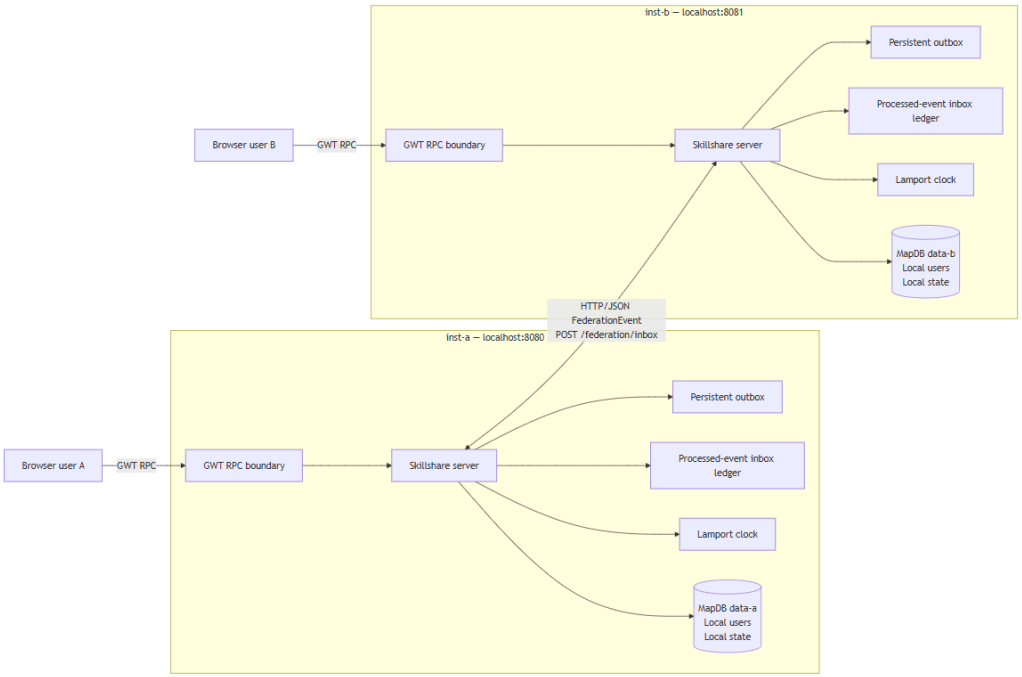

Figure 2: Federated Skillshare architecture. Each instance owns its local users and MapDB database, while cooperation is achieved through explicit HTTP/JSON federation events.

3 Design

The design follows a federated architecture. Each instance remains autonomous and persistent, while selected domain changes are propagated through explicit events. The goal is not to create one distributed database, but to coordinate independent instances through message passing.

3.1 Architecture

The architectural style is a small federated event-based web architecture. Browser clients use GWT RPC to interact with their local instance. Instance-to-instance communication uses HTTP `POST` requests carrying JSON federation events. There is no shared memory, shared database, broker, consensus service, or central coordinator.

This design addresses FR1, FR2, FR3, FR4, FR5, FR7, NFR1, NFR2, and NFR3. It keeps local responsibilities clear: an instance owns local users and local persistence, while federation only shares selected data required for the exchange community.

Data	Owner	Replication rule
Users and login data	Creating instance.	Not replicated; accounts remain independent per instance.
Profiles	User's home instance.	Not globally replicated; remote identity is represented by handle metadata.
Announcements	Origin instance.	Replicated to known peers for marketplace browsing.
Exchange requests	Announcement owner's instance.	Requester keeps a status replica; accept/reject events converge the view.
Chat messages	Both exchange participants.	Appended to both participant instances for accepted exchanges.
Reviews	Reviewed user's home instance.	Remote review is delivered home and used in local reputation.
Outbox and inbox ledger	Each local instance.	Not shared; they support delivery recovery and duplicate suppression locally.

Table 2: Replicated data and ownership boundaries.

3.2 Infrastructure

The final deployment contains `inst-a` and `inst-b`. The former is published at `http://localhost:8080`; the latter is published at `http://localhost:8081`. Inside the Docker network, each container reaches the other through its Compose service name on port 8080. The same application image is parameterized through `INSTANCE_ID`, `INSTANCE_URL`, `PEER_URLS`, and `DATA_DIR`.

Each service has an independent volume-mounted data directory. Docker Compose supplies a fixed list containing the other service URL. No discovery protocol, load balancer, broker, cache, or shared storage is introduced. The fixed topology is sufficient for a controlled demonstration but its broadcast cost and manual configuration do not scale to an open federation.

3.3 Modelling

The relevant domain entities are users, announcements, exchange requests, chat messages, reviews, and user reputation. A user is local to one instance. An announcement carries its origin instance. An exchange request records the requester and owner instances. Chat messages record the sender instance and Lamport timestamp. Reviews record both reviewer and reviewed identities.

The model distinguishes authoritative and replicated state. An announcement's origin instance owns its lifecycle. An exchange is authoritative at the announcement owner's instance while the requester retains a status replica. Chat is stored at both participant instances. Reputation is computed on the reviewed user's home instance from reviews stored there.

All federation messages share a `FederationEvent` envelope. The envelope contains an event ID, a type, an origin instance, and one feature-specific payload. The ID remains

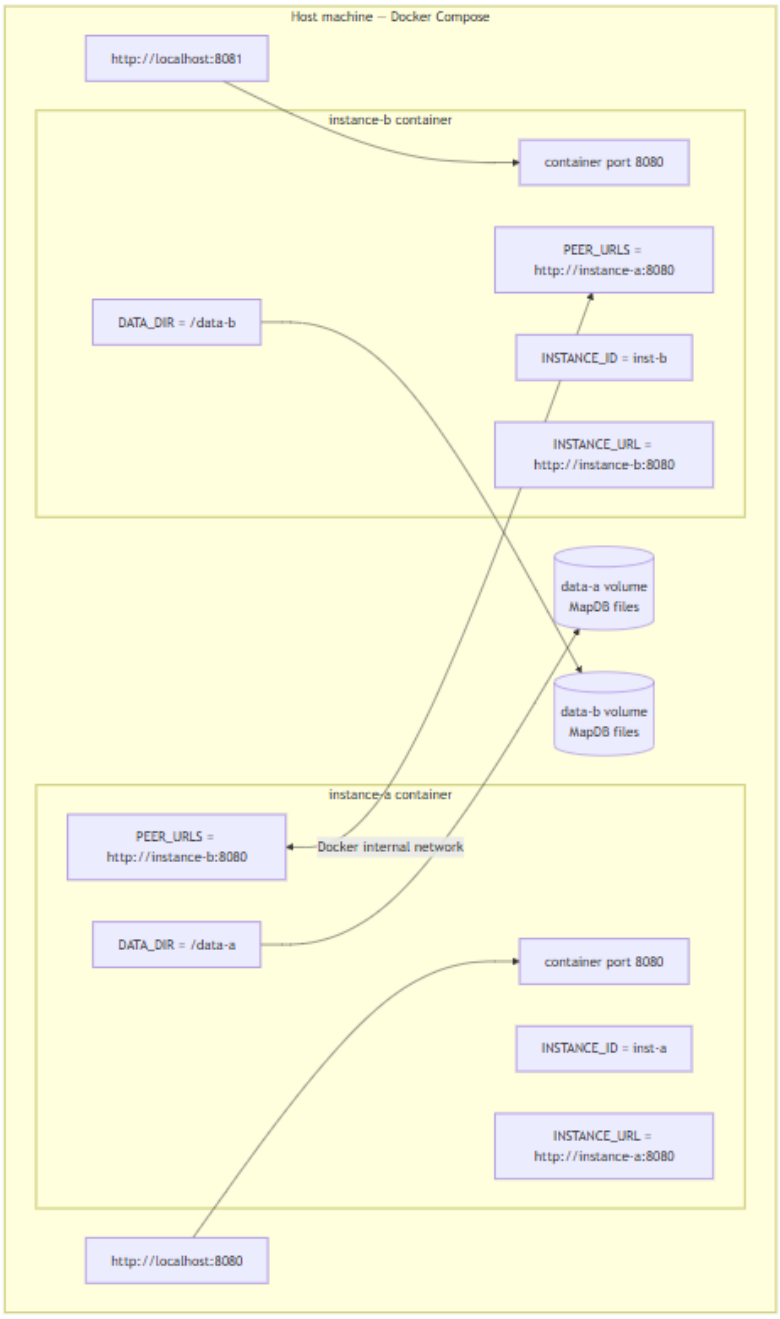

Figure 3: Docker Compose deployment for the local federation. The host exposes `inst-a` on port 8080 and `inst-b` on port 8081, while containers communicate through internal service names and keep separate data directories.

Event type	Payload	Meaning
`AnnouncementCreated`	Announcement	Create a marketplace replica.
`AnnouncementUpdated`	Announcement	Replace or create the latest received replica.
`AnnouncementDeleted`	Announcement ID	Remove a marketplace replica.
`ExchangeRequested`	Exchange request	Store a request on the owner's instance.
`ExchangeAccepted`	Exchange request	Propagate accepted status to the requester.
`ExchangeRejected`	Exchange request	Propagate rejected status to the requester.
`ChatMessageCreated`	Chat message	Append a message to the remote conversation replica.
`ReviewCreated`	Review	Store a review on the reviewed user's home instance.

Table 3: Current federation event types.

attached to the event when an outbox delivery is retried. The origin identifies the instance that produced the event.

3.4 Interaction

A normal federated write follows a local-first event flow. A manager validates the operation and updates a local repository. It then creates a domain event. The federation client creates one persistent outbox record per known peer and attempts an HTTP delivery. The peer inbox deserializes the event, checks whether it has already been processed, applies relevant state locally without rebroadcasting it, and records the event ID.

Receiving code deliberately uses repositories rather than managers that would broadcast again. This prevents an incoming event from circulating indefinitely. Directed exchange, chat, and review events currently share the broadcast path; receivers filter events based on instance metadata where applicable.

3.5 Behaviour

Each instance behaves as both a local web server and a federation peer. For local browser actions, the GWT RPC servlet delegates to manager classes. Managers enforce local business rules and update repositories. For remote federation actions, the inbox servlet receives JSON events and dispatches them to repository-level operations to avoid rebroadcasting.

Announcements are broadcast after local create, update, or delete. Exchange requests are created locally by the requester and then stored on the owner's instance. Accept/reject decisions are made by the owner and sent back so the requester replica converges. Chat messages are appended locally and then propagated to the other participant's instance. Reviews are validated and stored at the reviewed user's home instance, where reputation is computed.

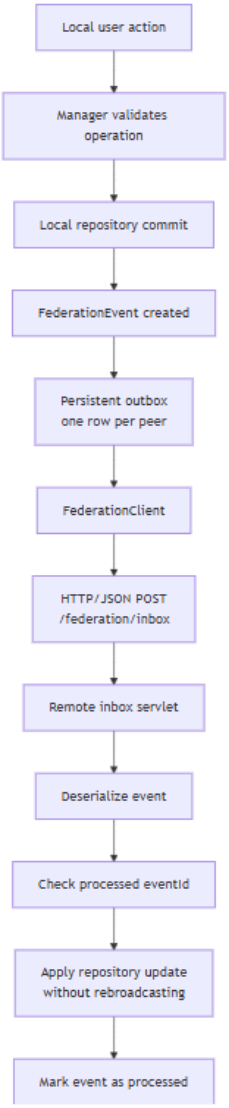

Figure 4: Local-first federation event flow. A local operation is committed first, then represented as a federation event, stored in the outbox, delivered through HTTP/JSON, applied by the remote inbox, and finally recorded in the processed-event ledger.

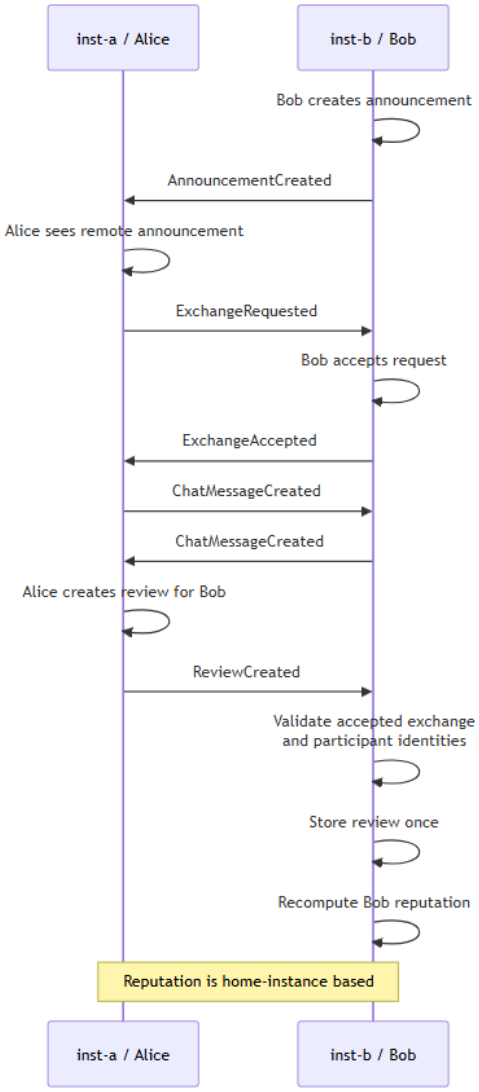

Figure 5: End-to-end federated scenario. A remote announcement leads to an exchange request, acceptance, federated chat, and a federated review that updates reputation on the reviewed user's home instance.

Federated chat uses Lamport clocks to represent causal progress without relying on synchronized wall clocks [4]. For a local send, the instance increments its logical clock and attaches the new value to the message. For a received message with timestamp r, the receiver updates its value L according to $L \leftarrow \max(L, r) + 1$. Messages are displayed in a deterministic total order using the tuple `(lamportTimestamp, senderInstance, wallClockTimestamp)`.

3.6 Data and Consistency Issues

Persistent data is stored in MapDB maps. This includes local domain data, replicated domain data, the outgoing federation outbox, the processed-event inbox ledger, and the Lamport clock. MapDB fits the existing application because the centralized project already used embedded persistent maps, and the federated design requires each instance to own a separate local file rather than a shared database.

The consistency model is eventual rather than immediate. Events are delivered asynchronously, so remote replicas may be stale while a peer is unreachable or while retry has not yet succeeded. The design avoids distributed transactions and consensus. Instead, it relies on ownership rules, local-first writes, event propagation, and receiver-side idempotency.

The current event model has no general entity version, tombstone mechanism, or formal conflict-resolution rule for all mutable events. Announcement updates are treated as replacement by the latest received representation, but concurrent updates and out-of-order mutable events are not fully addressed. Chat is append-only and ordered with Lamport timestamps, which solves the visible ordering problem for conversations but does not provide vector-clock concurrency detection.

3.7 Fault-Tolerance

Dependability is addressed through persistent failure information and recovery. A failed HTTP request does not cause the event to disappear. The outbox records its target, status, attempt count, timestamps, and error. A background scheduler and a manual endpoint can retry it. Integrity under retry is supported by event IDs, receiver-side validation, idempotent repository operations, and the processed-event ledger.

A Jakarta servlet exposes `POST /federation/retry` as a manual trigger. A lifecycle listener also starts a scheduled executor that periodically invokes retry. These mechanisms support the scenario in which one peer is stopped, the other accepts local work, and the missed events are sent after the peer returns.

Persistent retry gives the federation an at-least-once delivery model. A peer may receive the same event more than once, for example if the first application succeeds but the sender does not observe the response. The sender therefore does not try to infer exactly-once success from the network. Instead, every event carries a stable ID and the receiver maintains a persistent processed-event ledger.

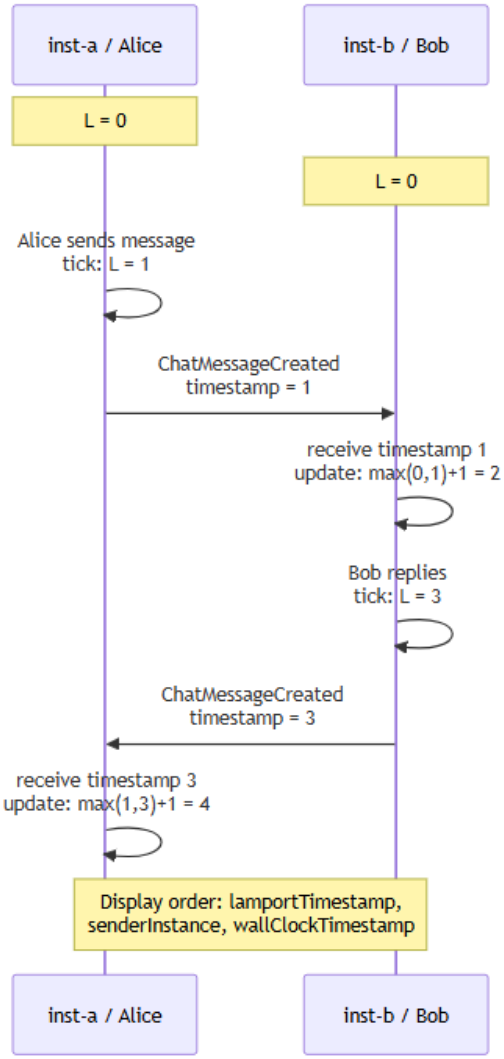

Figure 6: Lamport logical time in federated chat. Sending a message increments the local clock, receiving a remote message updates it with $\max(local, remote) + 1$, and both replicas sort messages deterministically using the Lamport timestamp and tie-breakers.

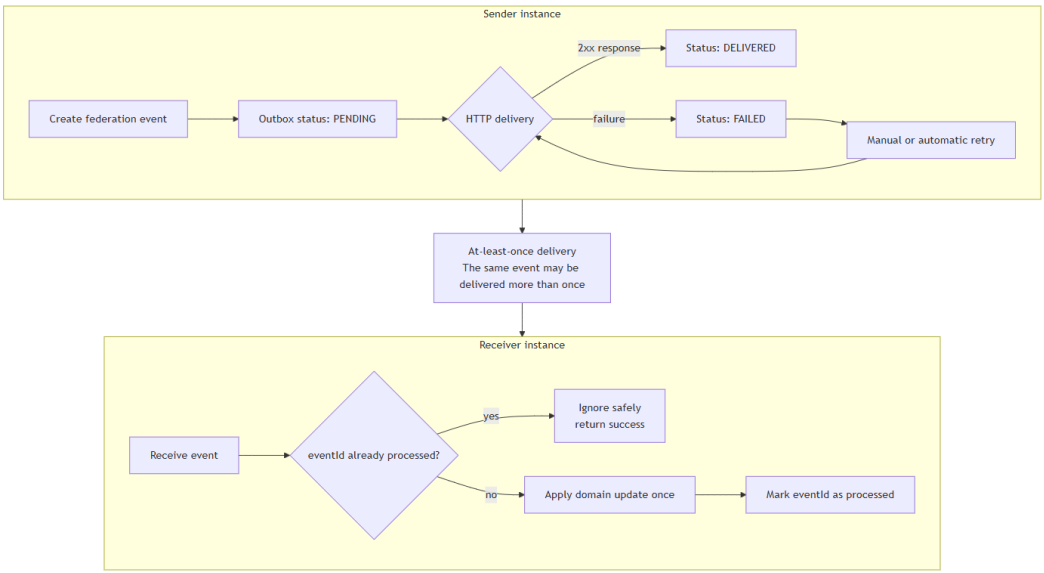

Figure 7: Persistent outbox, retry, and inbox idempotency. Sender-side retry gives at-least-once delivery, while receiver-side event IDs and the processed-event ledger prevent ordinary duplicate effects.

3.8 Availability

The design deliberately prioritizes local availability for the demonstrated operations. Managers persist valid local operations before federation delivery is known to have succeeded. If the peer is offline, the origin user still observes local success. The remote instance remains stale until recovery. This avoids making local service availability depend on the health of every configured peer.

There is no caching layer and no load balancer in the prototype. Marketplace replicas are not treated as a generic cache but as selected replicated domain state. During a network partition, each instance continues serving its own local users and retains outgoing events for later delivery. This is an AP-style choice for the specified local operations, not a formal proof that every operation of the application is AP.

3.9 Security

The web application retains local login and domain authorization rules. Users authenticate to their own instance, and accounts are local: a user created at `inst-a` does not automatically exist at `inst-b`. Federated identity prevents same-named users from being confused across instances.

Authorization is mostly domain-specific. A local user cannot manage another user's local announcements. A review can only be created for an accepted exchange by one of its participants. The reviewed user's home instance validates the event origin, target, exchange status, participant identities, rating, and duplicate rule before storing a

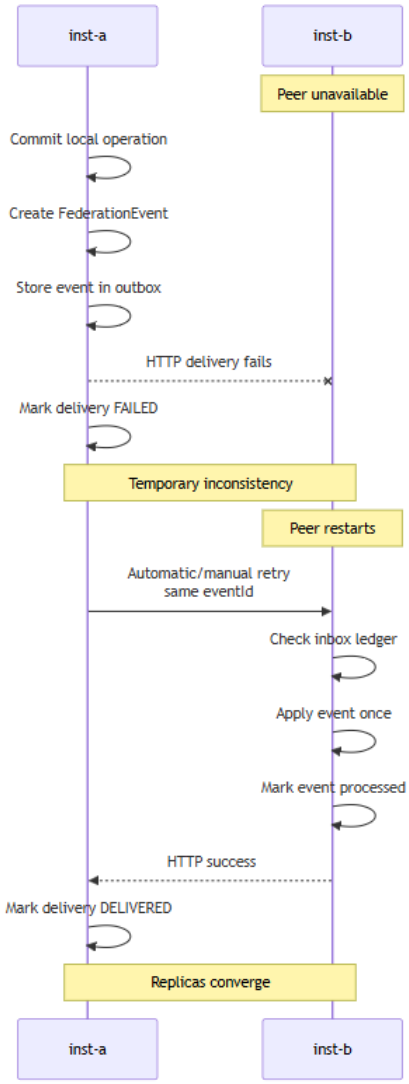

Figure 8: Offline peer recovery. The origin instance remains available and stores failed deliveries in the outbox; after the peer restarts, retry delivers the same event ID and the remote replica converges.

federated review.

The federation channel does not implement cryptographic peer authentication, message signatures, or TLS-based trust management. Structural checks on origin and participant fields do not prove that a network sender owns an asserted identity. The manual retry endpoint is intended for the controlled local demonstration rather than public exposure. Stronger authentication, authorization, replay protection, and trust configuration remain future work.

4 Implementation

The implementation preserves the existing Java/GWT application and adds a server-to-server federation layer. Local browser requests continue to use GWT RPC. Federation requests use explicit HTTP `POST` operations carrying JSON objects. JSON was chosen because it is human-readable, easy to inspect, and standardized [1]. The federation interface uses REST technologies and HTTP conventions, but it is not a full resource-oriented REST architecture in the sense of Fielding's architectural style [2].

Gson serializes and deserializes the `FederationEvent` envelope. Java's standard HTTP client performs synchronous server-to-server `POST` requests. MapDB stores local domain data and the federation support structures. The implementation deliberately avoids a message broker. This keeps the communication path visible for educational purposes: the application itself constructs the message, serializes it, sends HTTP, observes failure, persists delivery state, retries, and performs receiver-side deduplication.

The announcement federation implementation emits `AnnouncementCreated`, `AnnouncementUpdated`, and `AnnouncementDeleted` events. Exchange federation emits `ExchangeRequested`, `ExchangeAccepted`, and `ExchangeRejected`. Chat emits `ChatMessageCreated`. Reviews emit `ReviewCreated` and are used by the reviewed user's home instance when recomputing reputation.

4.1 Technological details

The project targets Java 17 and is built as a multi-module Maven application. The shared module contains RPC interfaces and serializable data transfer objects. The client module contains the GWT single-page interface. The server module is packaged as a WAR and runs on Jetty 12 using Jakarta servlets.

MapDB 3.0.10 stores persistent map collections. Every Docker instance receives a different `DATA_DIR`, which separates the database files. Gson 2.11.0 serializes federation events, while Java's standard HTTP client performs the HTTP communication. Docker Compose creates the two configured runtime instances.

The backend separates RPC or HTTP entry points, manager-level business rules, and repository persistence. Local browser requests enter through GWT RPC servlets. Federation requests enter through a plain Jakarta servlet because their caller is another server sending JSON, not a GWT browser proxy. Outbox, inbox, and Lamport state each have dedicated persistent collections.

5 Validation

Validation combines automatic tests and a controlled Docker scenario. Automatic tests exercise repository, manager, event, inbox, outbox, retry, and logical-clock behavior. Manual acceptance tests exercise the complete two-instance web application because some behaviours involve two browsers, two ports, Docker networking, and user interface state.

5.1 Automatic Testing

The server module uses JUnit 5 and Mockito. Repository and manager tests use an in-memory MapDB. The inspected repository currently contains 21 test classes and 176 methods annotated with `@Test`. This is a static source count, not a claim that the current full suite has executed successfully in the report environment.

Federation-specific tests cover event creation, outbox state, client delivery failure, retry scheduling, lifecycle integration, the retry servlet, the inbox ledger, duplicate inbox delivery, exchange filtering and propagation, chat participant identity, Lamport clock behavior, deterministic message order, and federated review validation and reputation.

The tests are run from the Maven project with `mvn test`. Build-level validation can also be performed with `mvn clean install`, which compiles the modules and the GWT client. The main rationale is to isolate distributed-system mechanisms that are otherwise difficult to observe manually: idempotency, retry state, delivery failure, and Lamport ordering.

5.2 Acceptance test

The acceptance validation is a two-instance Docker scenario. It is manual because the user interface, browser sessions, two running containers, and replicated state need to be observed together.

6 Deployment

The application requires Docker with Docker Compose. From the project `app` directory, the intended startup command is:

```
docker compose up -d --build
```

Compose builds one image and starts two services. `instance-a` publishes container port 8080 as host port 8080 and uses `INSTANCE_ID=inst-a`. `instance-b` publishes the same container port as host port 8081 and uses `INSTANCE_ID=inst-b`. Their peer URLs use the internal service names `http://instance-b:8080` and `http://instance-a:8080`. Separate host directories, `data-a` and `data-b`, are mounted as the database directory.

The services can be stopped with:

Scenario	Current evidence	Manual acceptance action
Announcement replication both ways	Unit-level federation tests	Verify both directions in Docker.
Peer offline and retry	Outbox and scheduler tests	Stop peer, restart it, and observe auto/manual retry.
Duplicate delivery	Inbox ledger tests	Replay one event in Docker.
Remote exchange request	Manager and inbox tests	Verify complete UI flow.
Accept/reject propagation	Automated status tests	Verify both outcomes across instances.
Federated chat	Manager and inbox tests	Verify two-browser conversation.
Lamport ordering	Clock and ordering tests	Capture equal replica order.
Federated review	Review and inbox tests	Verify remote review delivery.
Reputation update	Review manager tests	Verify home-instance UI.
Same username on both instances	Identity-aware unit tests	Verify UI labels and permissions.
Restart persistence	MapDB/clock unit evidence	Verify databases, outbox, inbox, and clock after restart.

Table 4: Validation status and required acceptance evidence.

```
docker compose down
```

Stopping containers does not remove the mounted databases. Deleting data directories is a separate destructive operation and is not part of the normal demo procedure. Expected outcomes are that `http://localhost:8080` serves `inst-a`, `http://localhost:8081` serves `inst-b`, and both instances can communicate internally through Docker service names.

7 User Guide

The intended federated demonstration proceeds as follows.

1. Start the two services with Docker Compose.
2. Open `http://localhost:8080` and register or log in as a user on `inst-a`.
3. Open `http://localhost:8081` in another browser profile or private window and register or log in as another user on `inst-b`.
4. Create a skill announcement on one instance.
5. Refresh the marketplace on the other instance and observe the owner displayed as `user@instance`.
6. From the remote instance, request the announcement.

7. On the owner's instance, accept or reject the request and observe the propagated status on the requester instance.
8. For an accepted exchange, open chat on both instances and exchange messages.
9. Leave a review for the remote participant.
10. On the reviewed participant's home instance, inspect reputation.

Accounts are local. An account created at port 8080 does not automatically exist at port 8081. Even if the same username is registered on both instances, the two identities remain distinct because their handles include the instance.

8 Self-evaluation

It should be noted that each student is only responsible for their own section.

8.1 Yeray González Menéndez

- **Implemented features.** Worked jointly on the full federated extension: the `FederationEvent` envelope with its Gson-based JSON serialization and the HTTP/JSON inbox endpoint, federated identity (`user@instance`), announcement federation, the exchange-request flow with accept/reject propagation, federated chat with its persistent Lamport logical clock, federated reviews and home-instance reputation, and the reliability layer composed of the persistent MapDB outbox, retry scheduler, manual retry endpoint, and processed-event inbox ledger.
- **Testing and validation.** Contributed together to the JUnit 5 and Mockito test suite covering the outbox, retry, inbox ledger and duplicate delivery, exchange propagation, chat participant identity, Lamport ordering, and federated review and reputation validation, as well as restart persistence.
- **Documentation and coordination.** Set up the Docker Compose two-instance deployment and co-authored the report and figures.
- **Strengths and weaknesses.** The main strengths are the clear federation boundary, local-first availability, persistent recovery, and explicit idempotency mechanism. The main weaknesses are the fixed peer list, limited security model, lack of general conflict resolution, and the local two-container scope.
- **Lessons learned.** Gained a shared understanding of the CAP trade-off through local-first writes, of why at-least-once delivery makes receiver-side idempotency essential, and of why logical time is needed for causal ordering across independent machines.

8.2 Jorge Vigil Bravo

- **Implemented features.** Worked jointly on the full federated extension: the `FederationEvent` envelope with its Gson-based JSON serialization and the HTTP/JSON inbox endpoint, federated identity (`user@instance`), announcement federation, the exchange-request flow with accept/reject propagation, federated chat with its persistent Lamport logical clock, federated reviews and home-instance reputation, and the reliability layer composed of the persistent MapDB outbox, retry scheduler, manual retry endpoint, and processed-event inbox ledger.
- **Testing and validation.** Contributed together to the JUnit 5 and Mockito test suite covering the outbox, retry, inbox ledger and duplicate delivery, exchange propagation, chat participant identity, Lamport ordering, and federated review and reputation validation, as well as restart persistence.
- **Documentation and coordination.** Set up the Docker Compose two-instance deployment and co-authored the report and figures.
- **Strengths and weaknesses.** The main strengths are the clear federation boundary, local-first availability, persistent recovery, and explicit idempotency mechanism. The main weaknesses are the fixed peer list, limited security model, lack of general conflict resolution, and the local two-container scope.
- **Lessons learned.** Gained a shared understanding of the CAP trade-off through local-first writes, of why at-least-once delivery makes receiver-side idempotency essential, and of why logical time is needed for causal ordering across independent machines.

9 Future works

The fixed peer list is the most visible scope reduction from the initial proposal. It makes the two-node demonstration reproducible, but instances do not discover peers dynamically. Events also use a simple broadcast-to-known-peers strategy rather than ActivityPub routing. The system does not implement actor documents, WebFinger, ActivityPub inbox/outbox semantics, or protocol compatibility and must not be described as a full ActivityPub server.

The demonstration runs two containers on one host. It exercises autonomous processes, separate databases, message passing, and peer unavailability, but not real Internet latency, independent hosts, or a large federation. The security model is also incomplete: federation events have no signatures or peer authentication.

Lamport clocks cover chat only. Vector clocks are absent, so concurrency cannot be detected precisely. Other mutable event families have no general version, tombstone, or conflict-resolution mechanism. Consensus, quorums, and distributed transactions are intentionally absent.

Future work includes dynamic peer discovery, selective routing, signed and authenticated events, TLS-based trust configuration, schema and capability versioning, dead-letter handling, bounded exponential backoff, atomic inbox application, message-level deduplication, stronger conflict resolution, and deployment across independent hosts. A production design would also need moderation, abuse prevention, privacy policy, observability, and database migration support.

References

[1] Tim Bray. The javascript object notation (json) data interchange format. Technical Report RFC 8259, Internet Engineering Task Force, 2017.

[2] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[3] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.