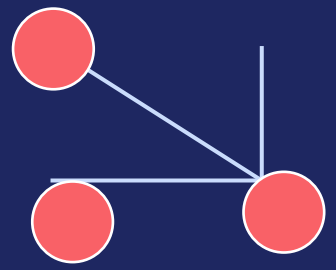


Skillshare

A federated peer-to-peer skill-exchange platform

Distributed Systems
Università di Bologna — Informatica per il Management
Jorge Vigil Bravo – Yeray González Menéndez

The domain: a barter economy of skills

No money — the unit of value is a reciprocal exchange

Dual-role users

Exchange requests

Private chat

Reviews & reputation

Architecture and stack

One self-contained Java web application — which is what makes federation possible

GWT UI compiled to JavaScript, runs in the browser

Serializable types crossing the GWT-RPC boundary

Services, domain managers, MapDB repositories

GWT 2.13 | MapDB | Maven | JUnit 5 | Docker Compose | Gson (federation)

Each instance is completely self-contained: its own embedded transactional database, no external services. An instance needs nothing but itself to run — the precondition for running many of them as peers.

The base application

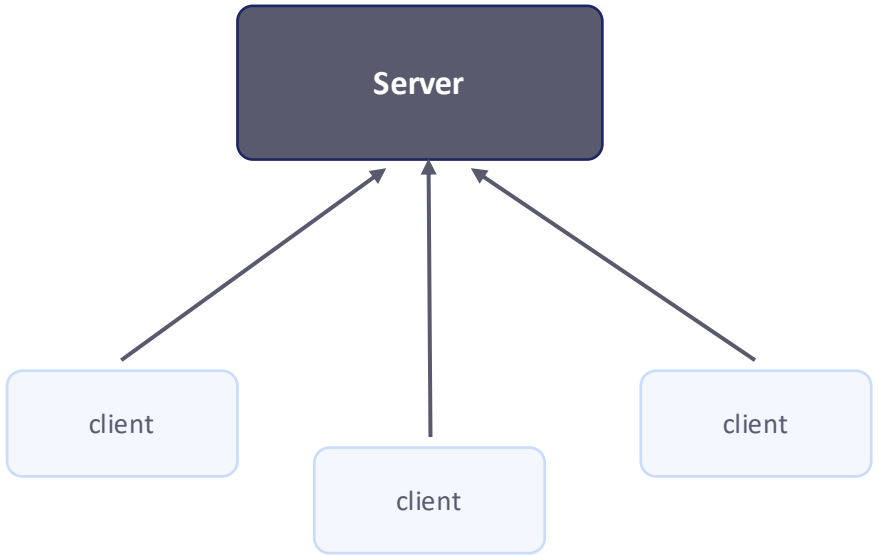
The full collaboration lifecycle, end to end

1. **Accounts** — Register and sign in; clear, readable error feedback
2. **Marketplace** — Create, search, edit and delete announcements
3. **Exchange requests** — Request on someone's announcement; owner accepts or rejects
4. **Private chat** — Opens between the two participants once accepted
5. **Reviews & profiles** — Rate the exchange; public profiles and reputation

Why federation? Removing the central server

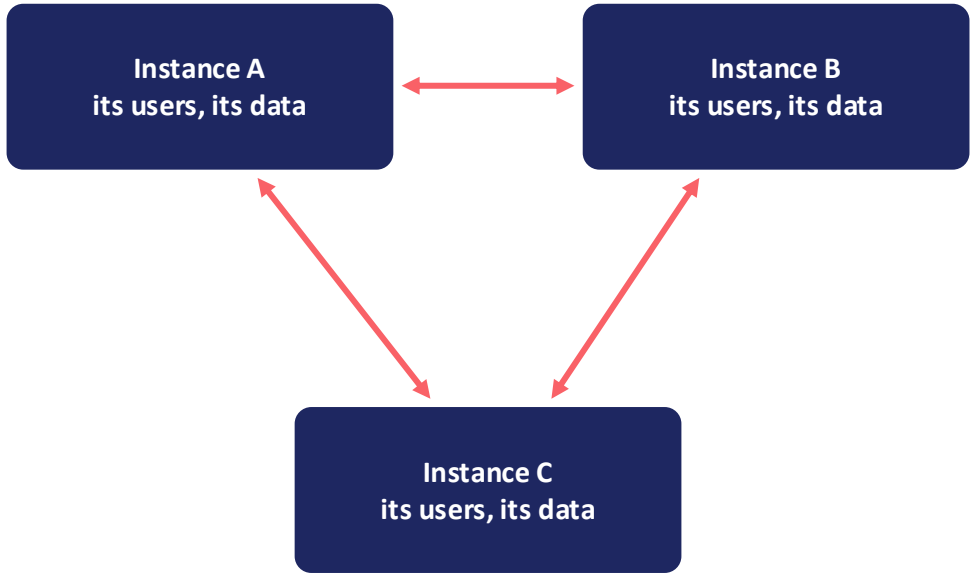
The ActivityPub / Mastodon model: a network of independent instances

One central server



single point of failure and control

A network of instances

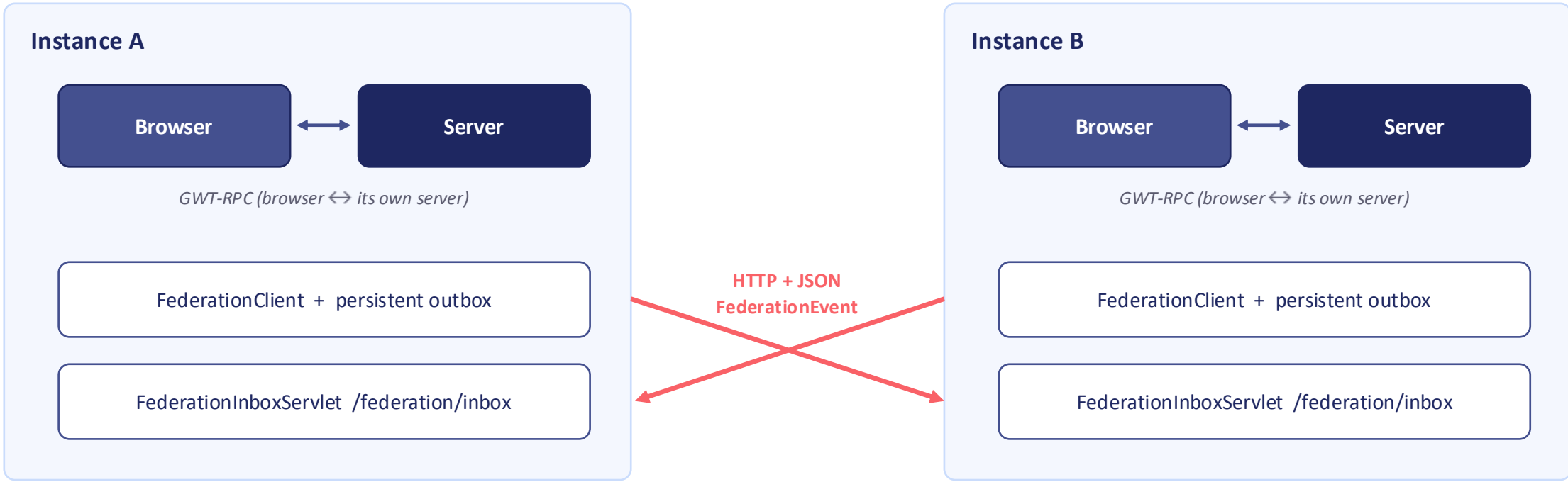


peers cooperate by messages — no shared memory, partial failures

A user on instance A can browse and request an exchange from a user on instance B, as if on the same site.

Federation architecture: domain events between peers

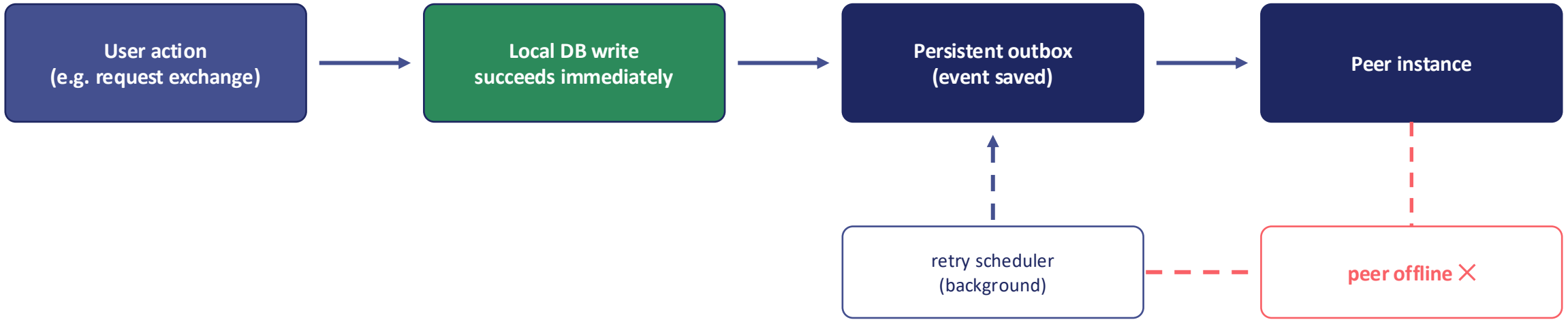
Instances exchange what happened — not database rows



AnnouncementCreated / Updated / Deleted · ExchangeRequested / Accepted / Rejected · ChatMessageCreated
Configured from outside (INSTANCE_ID, INSTANCE_URL, PEER_URLS) — the same code runs as any instance.

Design choice: availability over consistency

Local-first writes; peers converge eventually — a deliberate point on the CAP trade-off



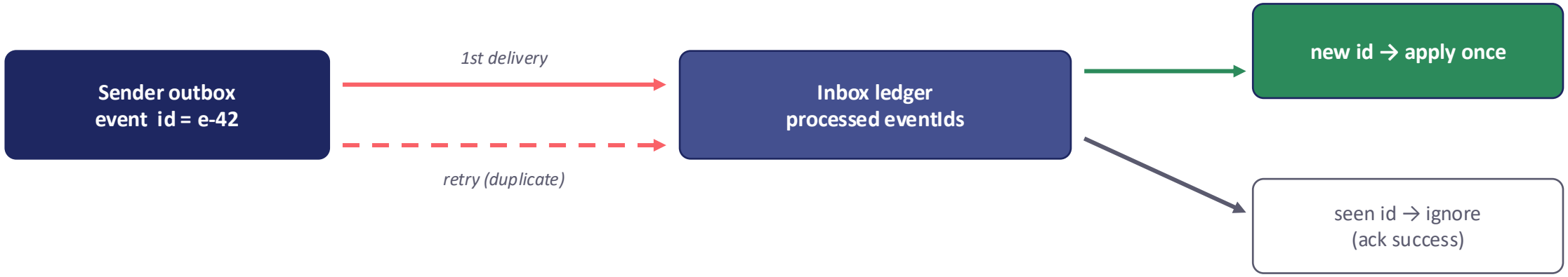
delivery re-attempted when the peer returns — the local action was never blocked

The system always responds locally — a user's action never fails because a peer is down

A partition degrades freshness, never availability — replicas converge to the same state eventually

At-least-once delivery needs idempotency

Deliver at least once — apply at most once



Why retries force this: the persistent outbox re-sends until acknowledged, so the same event can arrive more than once — the standard at-least-once model.

Every FederationEvent carries a unique eventId. The inbox checks a persistent ledger before applying; failed applications are not marked processed, so they can still be retried.

Chat is the critical case: messages are append-only with no natural key — without the ledger, a retry would duplicate the message.

Federated identity: user@instance

Usernames are only unique within an instance — like e-mail or Mastodon handles

- **Announcements —** carry an originInstance — their home instance is part of their identity
- **Users —** carry their home instance; the UI shows handles like carol@inst-b
- **Permission checks —** compare username AND instance: a remote “alice” can never edit the local Alice's announcement
- **Self-request rule —** instance-aware too: alice@inst-a may request alice@inst-b's announcement, never her own

Federated workflows

Same pattern everywhere: act locally, let events carry the change, rely on idempotency

Announcements

replicated to every peer

the marketplace is global:
everyone sees everyone's offers

Exchange requests

directed to the owner's instance

authoritative on the owner's side;
accept/reject propagates back

Chat messages

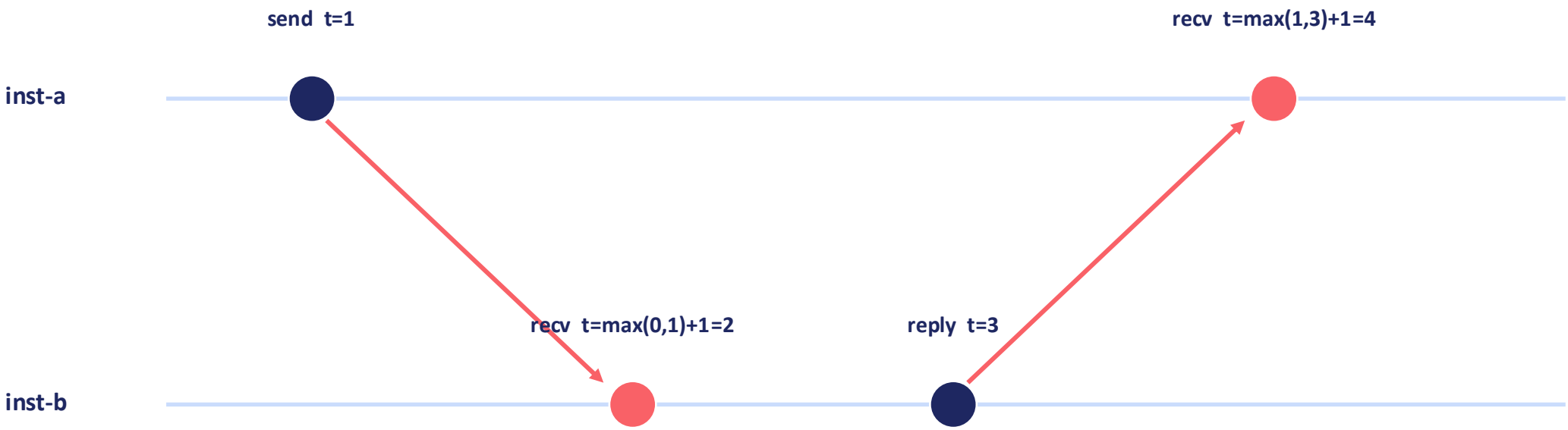
sent to the other participant

both replicas converge to
the full conversation

Cross-instance exchange, end to end: alice@inst-a requests on bob@inst-b's announcement → event stored on B under “Received” → Bob accepts → status event returns to A → Alice's replica flips to ACCEPTED → chat opens across the two instances.

Logical clocks: one order for every replica

Lamport timestamps give the chat a deterministic total order — no synchronized wall clock



happens-before preserved: a reply always carries a strictly greater timestamp than the message it answers

tick() on every local send · **update(received) = max(local, received) + 1** on every receive · persisted in MapDB (survives restarts)

Concurrent messages (equal t) break ties by instance id → order by **(lamportTimestamp, instanceId)** — a total order every replica agrees on: both instances render the identical conversation.

The eventual-consistency picture

Three pillars — and honest, scoped limitations

Availability	Idempotency	Ordering
local-first writes, persistent outbox, automatic retry	unique eventId, persistent inbox ledger, apply at most once	Lamport logical time, deterministic tie-break, identical replicas

Known limitations — documented as design decisions, not silent gaps

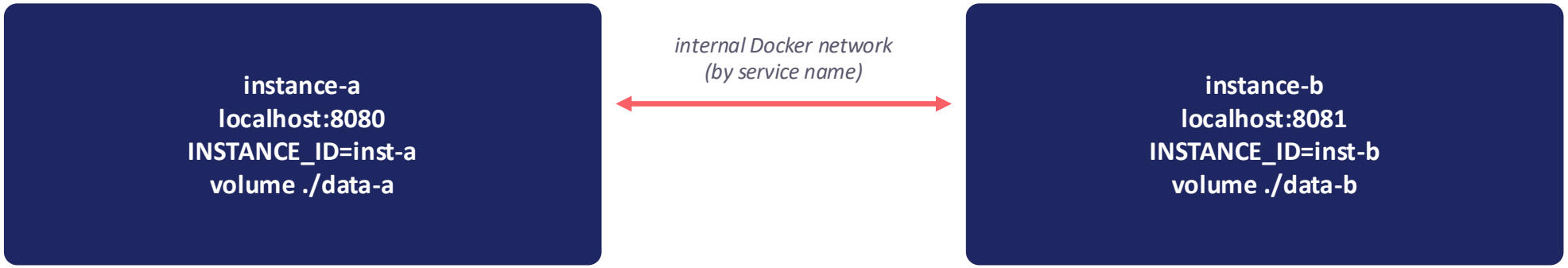
Announcements rely on single-writer semantics — only the owner, on its home instance, ever edits one — so there are no concurrent writes to reconcile. Residual risk: out-of-order delivery from that single writer; a Lamport last-writer-wins rule is the natural extension.

Deletions have no tombstones yet — a heavily delayed update could resurrect a deleted announcement; timestamped tombstones are future work.

Profiles and reviews are per-instance — federating them is the next increment.

Deployment: a two-node distributed system on a laptop

Docker Compose, isolated data volumes, internal network



Live demo

- Request an exchange from A on a B-owned announcement → it appears on B under Received
- Accept on B → the status flips back to ACCEPTED on A
- Chat across instances → both sides show the identical, Lamport-ordered conversation
- Stop instance B, act on A (still works) → restart B → the retry delivers the pending events

Engineering practices

The Software Engineering side of the same codebase

170+ automated tests

JUnit 5; features confirmed only after tests and a Docker smoke-test pass

Isolated by design

in-memory MapDB test mode + a capturing federation client — tests never touch the network

Strict GWT layering

every class is client / shared / server; violations fail only at runtime, so the boundary is policed

Honest process artifacts

report and diary describe what actually happened — trade-offs and limitations included

Conclusions

What we built

- A working P2P skill-exchange platform
- …turned into a federated network of instances
- Available under failure (outbox + retry)
- Correct under retries (eventId idempotency)
- Ordered across replicas (Lamport clocks)

What we learned

- There is no global clock you can trust
- At-least-once delivery forces idempotent design
- Availability vs consistency is a choice you make on purpose
- Identity must be qualified: user@instance

Future work: federated profiles and reviews · timestamped tombstones for deletes · Lamport last-writer-wins for announcements

Thank you — questions welcome