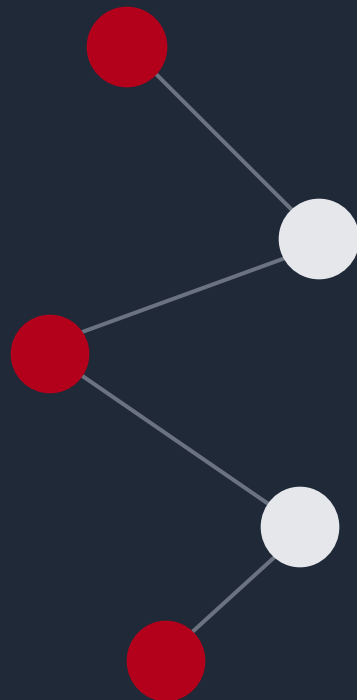


ConverGO

A decentralized DVCS based on CvRDT updates

Distributed Systems Course 2024/2025
Stefano Guidi



■ What was built

A proof-of-concept peer-to-peer version control system in Go, where replicas converge automatically, without merge conflicts.

DVCS

- Decentralized by design
- Replicated file history
- Every peer stores the files locally
- Causal history of operations

CRDT

- State-based CRDT wrapping WOOT
- Deterministic, idempotent merge
- Eventual consistency by design

Collaborative

- Concurrent edits to shared files
- Push-based broadcast between peers
- Consistency is the key goal

CLI app · Cluster-based · decentralized · ~90 hours

■ DVCS vs CRDT: the idea

Classical DVCS

- Explicit sync and merge
- Manual conflict resolution
- Commits, deterministic history
- Assumes intermittent coordination



CRDT-based systems

- Automatic convergence
- Replicas evolve independently
- Eventual (weak) consistency
- No coordination on edits

ConverGO: a single convergent history of operations shared by the cluster, exploring the tradeoffs of causality, delivery semantics and scalability.

■ Main design choices

Push-based

Peers broadcast their state; no pull

Operations, not commits

History = individual file operations

One CRDT per file

Each file has its own history

At-most-once delivery

No retransmission, loss is tolerated

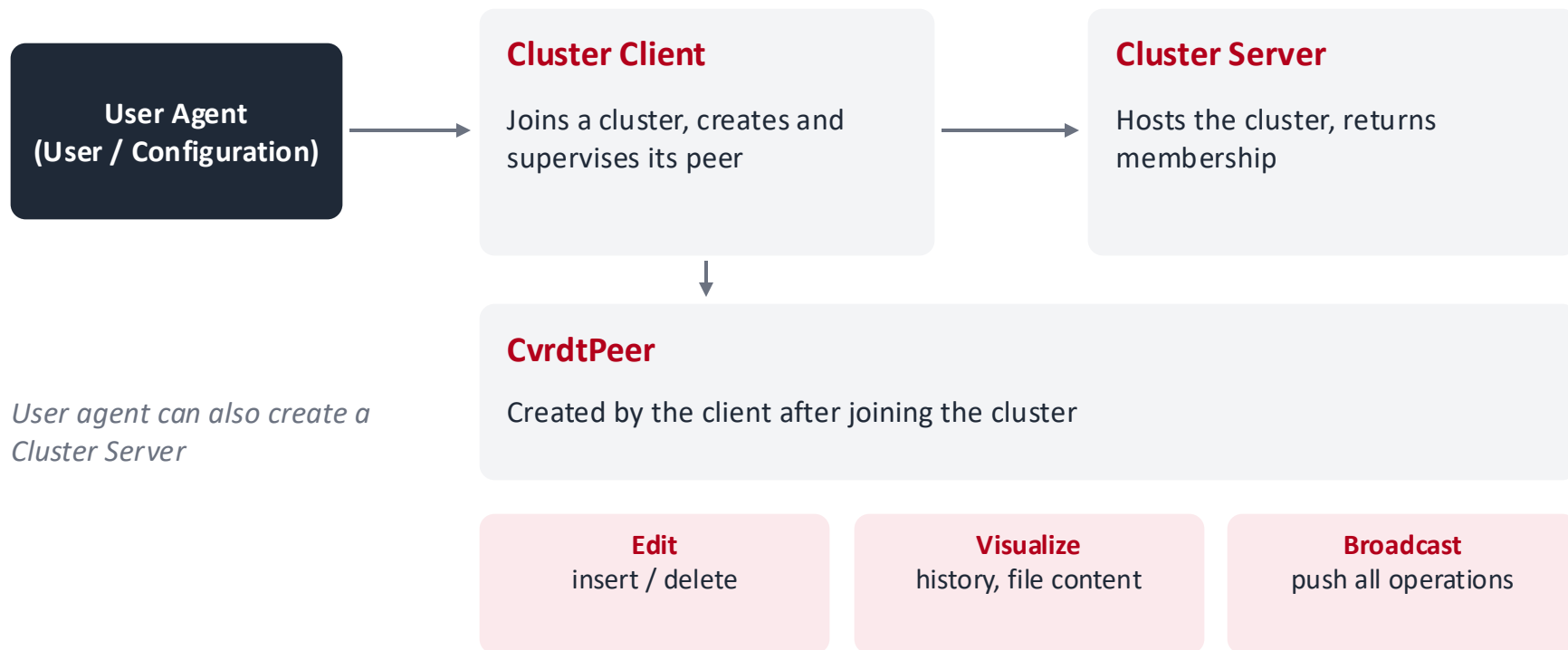
Central Cluster Server

Single source of truth for membership

CLI + Docker

Interactive and automated modes

■ System use cases



■ Requirements

Non-functional

- **Eventual consistency of CvRDT peers**
- Strong consistency of cluster membership
- Fault tolerance to peer failures
- Software portability
- Robustness to communication loss

Functional

- Join a cluster, get peer id and members
- Connect to peers, propagate full CRDT state
- Receive a state and merge it (CvRDT merge)
- Query the server before pushing
- Create / modify / delete file content
- **View causal partial order graph of operations**

■ Distributed system features

Eventual consistency

- CvRDT merge is deterministic, associative, commutative, idempotent
- Any exchange → convergence

At-most-once

- Full state is re-sent, so a lost message only delays convergence
- No retransmission, no duplicate tracking

CAP, two views

- Server queries: consistency over availability
- Peer to peer: eventual consistency

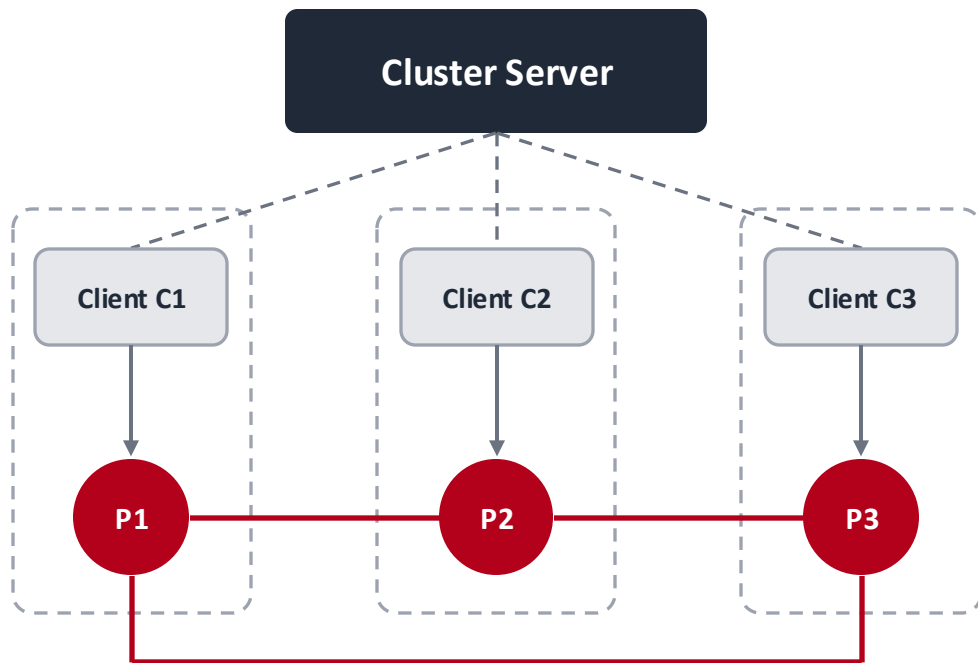
Not addressed: authentication, large-scale deployments, memory efficiency (state grows with operations and peers).

■ Layered architecture

Command	CLI loop, parser, command handlers
Cluster	ClusterNodeStore, client and server logic, logging
CvRDT peer	Repository, CRDT replicas, history, push
Session	Topics, UUID sessions, timeouts, drops
Transport	TCP, handshake, header codec

Presented from user-closest to network-closest; not a strict layered architecture. Cluster + Cvrtd peer = Application layer

■ Infrastructure and topology



Dashed lines: membership with the server · Red: state push between peers · Dashed box: one application (client + peer)

Events handled by each node

Cluster Server

- PING JOIN: register peer, assign id
- PING RETRIEVE: return members
- PEER EXIT: remove departed peer

Cluster Client

- PEER CONNECTION: server link up
- PEER EXIT: server lost

CvrdtPeer

- PUSH: merge received state
- PEER CONNECTION: send state to it
- PEER EXIT: log only

■ Networking subsystem

Transport

- TCP implementation
- Handshake exchanges
PeerHostInfo
- Length-prefixed frames, 4 MB max

JSON Serialization

Session

- MessageBroker with topic handlers
- One session per exchange, UUID ids
- Drops on timeout, out-of-order, full queue

PeerMessage

PeerMetadata

id · name · address
topic · session · seqNo

Payload

opaque to lower layers,
decoded by the application

Envelope splits transport metadata from payload

■ Repository subsystem



File output

- StatePollingDumper writes the text snapshot to the node file system
- Only the state is dumped, not the operations

History

- Built from the same CvRDT state
- One history per file, rendered as linearization or DAG

■ Join and broadcast push

Cluster join

1

Client handshake with server

2

PING JOIN with client + peer address

3

Server validates, returns new id + members

4

Peer created, connects to members

Broadcast push

1

Ping server
get current members

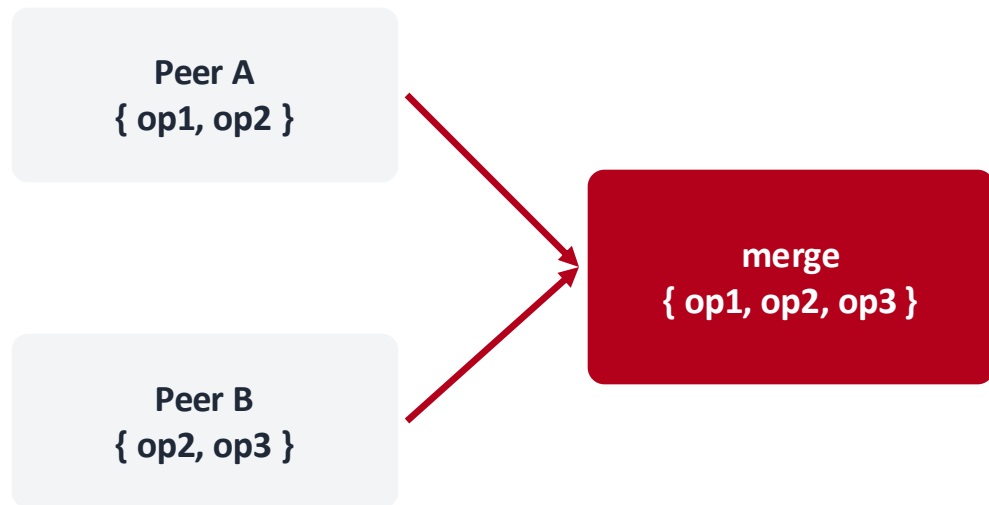
2

Align peers
connect new, drop departed

3

Broadcast
send all file states, receivers merge

■ CvRDT wrapping an operation-based CRDT



Delta-state CRDTs could avoid sending the full state.

Tradeoffs

- State = set of all received operations
- Merge adds only unseen operations
- No retransmission needed
- Cost: full state sent every time, so traffic grows with operations

■ WOOT: collaborative text CRDT

W-Character $\langle id, \alpha, v, idcp, idcn \rangle$

Operations

- $ins(a \prec e \prec b)$
- $del(e)$: tombstone, $v = false$
- Stable ids, not positions

Causal delivery

- Insert needs prev and next present
- Delete needs the target present
- Otherwise kept as pending

My modification

- $id = \langle \text{site-id}, \text{vector clock} \rangle$
- Lamport clock replaced to separate concurrent from causal events

■ Insertion integration and vector clocks

IntegrateIns(c, cp, cn)

- Take subsequence between cp and cn
- Empty: insert c directly
- Otherwise find competitors d0...dm
- Order with id tie-break, narrow interval
- Recurse until one gap remains

Lamport clock

- Smaller value does not imply causality
- Concurrent events may look sequential

Vector clock

- Exact happened-before and concurrency
- Size grows with replicas, no GC

History visualization

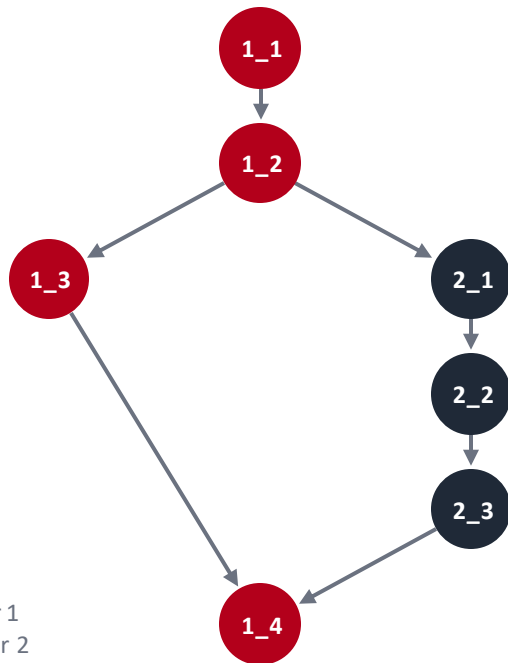
ASCII DAG, complex test

```
DAG: 7 operations, 7 causal edges
1_1 [Insertion] 'ThisTextWillStay' →1_2
└ 1_2 [Insertion] 'ThisTextWillBeRemoved' →1_3, 2_1
  └ 1_3 [Insertion] '))' →1_4
    └ 2_1 [Deletion] 'ThisTextWillBeRemoved' →2_2
      └ 2_2 [Insertion] 'Prefix:' →2_3
        └ 2_3 [Insertion] '(' →1_4
          └ 1_4 [Insertion] '!!!' (2 parents: 1_3, 2_3)
```

Two views

- Linearization: sort by id-clock pair
- DAG: happened-before, layered by depth

Causal DAG, complex test



■ Validation strategy

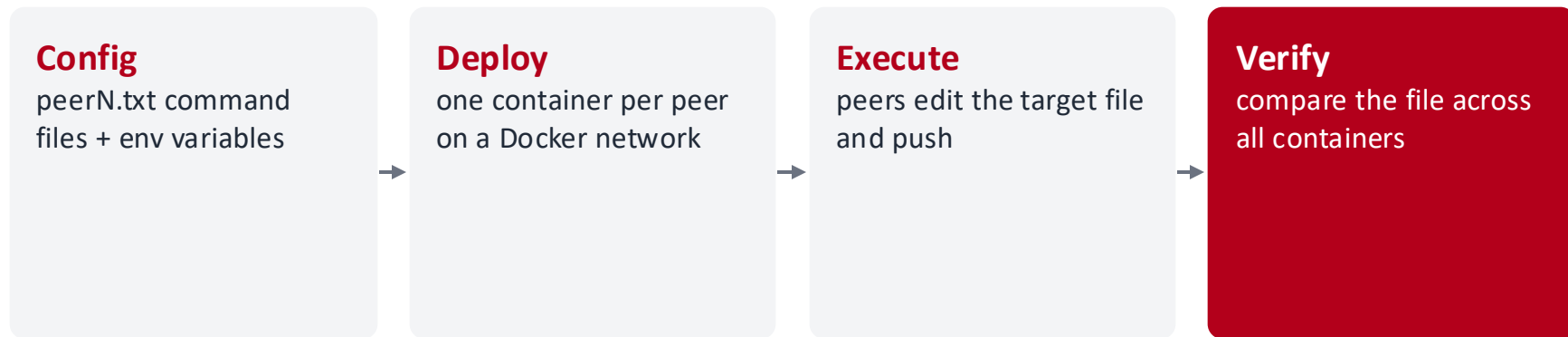
Unit and integration tests

- Peer and communication
- CvRDT and WOOT merge
- File contexts and polling
- History and DAG building
- Repository (files + history)
- Run with make test

Convergent testing

- End-to-end acceptance tests
- Simulated cluster of Docker containers
- Checks eventual consistency and the non-functional requirements
- Users can define their own scenarios

■ Convergent testing in practice



```
make dock-test N=3 TEST_DIR=three_peer_convergence
```

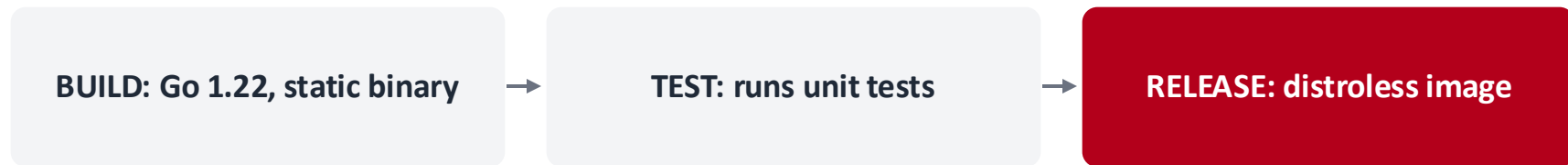
N is the maximum number of peers: fewer peers can run the same scenario. Identical files = convergence.

■ Acceptance scenarios

Scenario	Peers	Outcome	Validates
pairing	2	Late joiner receives state	Eventual consistency
three peer	3	Chained ops, all converge	Robustness to loss
complex	2	Concurrent edits, insert + delete, merged	Eventual consistency
peer left	3	Leaving peer does not block others	Fault tolerance

Cluster-membership consistency is validated implicitly by all scenarios.

■ Deployment



Peer Environment

- Container with hostname and Docker network
- Hosts one client and/or server
- Docker only, no Compose

Two modes

- Configuration: automated, detached containers
- Interactive: manual CLI exploration
- Makefile: dock-build, dock-net, dock-clean

■ Limitations

Server as single point

Needs consensus or proactive notifications

Ever-growing state

Full state sent, 4 MB message cap

Metadata growth

Vector clocks and tombstones, no GC

Ping before every push

Heartbeat could reduce load

No pull, no auth

Push-only, forgeable identities

File-level history

No repo-wide history, no commits

Conclusions

- CRDTs make DVCS convergence automatic, with no merge conflicts
- Convergence verified end to end across containers
- Tradeoffs exposed: causality, delivery, state growth, centralization
- Next: delta-state sync, decentralized membership, repo-wide history

Thank you.