

ConverGO - A DVCS based on CvRDT updates

Final Report for the Distributed Systems Course 2024/2025

`stefano.guidi10@studio.unibo.it`

September 26, 2026

This work presents the design and evaluation of a decentralized Distributed Version Control System (DVCS) based on State based Conflict-free Replicated Data Type (CvRDT). The project provides a proof-of-concept about how distributed collaborative approaches can be integrated along DVCS into a peer-to-peer version control environment while preserving eventual consistency and minimizing coordination overhead.

Traditional distributed version control systems rely on explicit synchronization and merge procedures to reconcile concurrent modifications performed by multiple users. While these mechanisms provide strong traceability and deterministic history management, they often require manual conflict resolution and assume intermittent coordination between replicas (this is sometime an intended and desirable behavior, especially when human control is necessary). In contrast CRDT-based systems are designed to guarantee weak-consistency (eventual consistency) automatically, to enable replicas to evolve independently while reaching a single consistent global state.

Through ConverGO, we want to demonstrate how CRDT techniques can be successfully used to improve decentralized DVCS systems while also exposing challenge and tradeoff about event causal relations, history management, semantic delivery and system scalability.

The resulting software is not to be intended as a real-world production ready library but as a proof-of-concept prototype to provide insight about the applicability of CRDT models in the DVCS context along discussion about its inner limitation and necessary improvoment to make it a working product.

Disclaimer

During the preparation of this work, the author used generative AI tools to improve the writing and presentation of the report and to assist in generating a portion of the unit tests, with the aim of improving code coverage.

All AI-generated content was reviewed and edited by the author as necessary. The author takes full responsibility for the final content of this report and for all changes made to the source code.

As discussed in the abstract, the developed software artifact is intentionally limited in scope, both in terms of its distributed-system characteristics and domain features. These limitations stem from the inherent complexity of developing a complete production-ready system, the decision to focus on specific distributed systems challenges, and the project's approximate 90-hour development constraint.

Nevertheless, it was considered essential to explicitly document these limitations, associated design trade-offs, and possible future improvements throughout the report. Some of these issues may be regarded as time-constrained technical shortcomings related to distributed systems implementation, while others represent deliberate architectural or design decisions. These considerations are discussed in detail in the Limitation section 10 and are referenced throughout the report where relevant. The reader is encouraged to read them as referenced or thoughtfully from the start, as they are a fundamental part of the report.

1 Concept

The product is delivered as a CLI application and can be used in two different modes:

- **Automated configuration mode:** This mode is primarily intended to support system testing by automatically building and configuring an infrastructure composed of multiple peers. Its main purpose is to verify the consistency of the system as a whole.
- **Interactive mode:** This mode allows the user to interact with other peers through one of their own. It is primarily intended for manually exploring and testing the application's functionality.

While both modes are fully functional from a practical standpoint, the configuration mode provides a simpler and more customizable way to evaluate and explore the system (it's also the chosen mode to validate the software requirements section 5.2).

The main objective is to build a decentralized Distributed Version Control System (DVCS). Rather than explicitly managing merge conflicts, Conflict-free Replicated Data Types (CRDTs) will be used to provide a unique, convergent sequence and history of operations.

The following aspects characterize the proposed system:

- The system will provide its services as a distributed cluster, in which each peer broadcasts its currently known operations to the other peers in the cluster.
- The system will follow a push-based communication model: peers can broadcast their data to other peers, but cannot explicitly request or pull data from them (read more about limit. item 11).

- Each cluster file that is created or edited will be stored in the local file system of each cluster member.
- Unlike a classical DVCS, the system will not introduce the concept of commits. Instead, the history will be composed of individual operations, such as a specific modification to a file (read more about commits vs operation tradeoff limit. item 17).
- The system will provide a unique, convergent history of operations that is shared across the cluster.

Although the system is decentralized from the perspective of the peers, whose primary responsibility is to exchange their currently known operations, a centralized component is required to maintain authoritative information about cluster membership. This component, referred to as the **Cluster Server**, acts as the single source of truth for cluster membership.

This represents a significant limitation of the system from a distributed-systems perspective. A detailed discussion of this limitation, along with possible improvements, is provided in the limit. item 4.

Furthermore, the creation and management of peers (**CvrdtPeers**) is handled by a client component, referred to as the **Cluster Client**.

The system use-case scenario can be described by the following fig. 1. Some important aspects to note are:

- The three main peer use cases are: creating an operation (i.e., editing a file that will be tracked by all peers in the cluster); visualizing these operations in terms of their history or providing the result of these operations (i.e., the resulting text) and broadcasting all currently owned operations to the other peers.
- A peer (**CvrdtPeer**) can be created only after the **Cluster Client** has been created and has joined an existing Cluster by connecting to a **Cluster Server**.
- The **Cluster Server** can be created locally by the same user agent that hosts the **Cluster Client**, or it can be an already existing remote server.
- The diagram highlights three fundamental infrastructural entities: the client, which supervises the peer and connects to a cluster (**Cluster Client**); the server, which hosts the cluster (**Cluster Server**); and the peer, which actively participates in the DVCS (**CvrdtPeer**).
- The two specializations of the User Agent role (Configuration and User) highlight the two underlying usage modes described at the start of the section, which still access the system in the same way.

The distributed and decentralized nature of this software provides two main advantages:

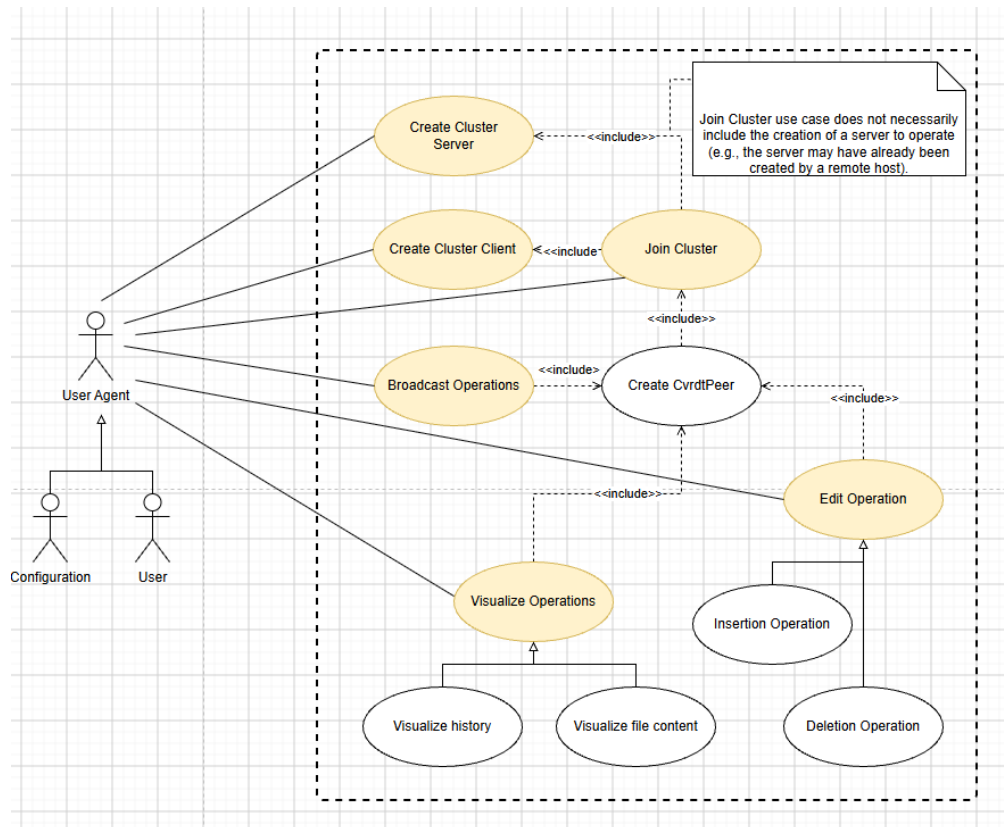


Figure 1: Use cases for the system

- Replication of the operation history for all edited files.
- The possibility of sharing data between different user in the cluster

More specifically, one of the focal distributed system quality attributes will be eventual consistency. This feature will be fundamental and will have a significant impact on fault tolerance, the CRDT approach, and the system's delivery semantics (see Relevant distributed System features section 2.1).

2 Requirements Elicitation and Analysis

Non-Functional Requirements

1. **Eventual Consistency for CvRDT Peers:** The system shall guarantee eventual consistency. Given sufficient time and in the assumption of eventual exchange between peers, every peer in the cluster shall converge to the same file state.
2. **Strong Consistency for Cluster Membership:** Every request to the cluster server shall return an up-to-date and consistent view of the peers currently joined to the cluster.
3. **Fault Tolerance to Peer Failures:** An unexpected peer shutdown or failure shall not affect the operation or availability of the remaining peers in the cluster.
4. **Software Infrastructure Portability:** The system shall be portable and configurable across multiple operating systems and deployment environments.
5. **Robustness to Communication Loss:** Temporary data loss, message loss, or communication failures between peers shall not compromise the system's consistency guarantees or other quality attributes.

Functional Requirements

1. A user shall be able to connect to join a cluster, obtain information about currently connected peers and receive its assigned cluster peer identifier.
2. A peer shall be able to establish connections with other peers in the cluster and propagate its complete CRDT state to them.
3. A peer shall be able to receive a CRDT state from another peer and merge it using the CvRDT merge algorithm.
4. Every peer shall be able to query the Cluster Server to obtain information about other peers before initiating to propagate its state.
5. A user shall be able to create, modify, or delete files content within the cluster-managed directory, generating the corresponding CRDT operations.
6. A user shall be able to visualize a causal graphical representation of the current CvRDT operations maintained by the peer .

2.1 Relevant Distributed System Features

In the following section, the distributed system features will be introduced. Some of them can be directly mapped to specific non-functional requirements (e.g., eventual consistency req. 1 and peer fault tolerance req. 3), while others need to be correctly implemented to support the functional requirements (e.g., network-based cluster communication for req. 1, req. 2, and req. 4).

The distributed aspects of the system are abstracted behind a *Transport* layer, which hides the following concerns:

- The underlying transport protocol: the effective implementation for communication between both CvRDT peers and the Cluster Server.
- Remote peer management and message delivery mechanisms.
- Basic serialization and deserialization strategies. Payload-specific serialization and deserialization are handled at a higher layer.
- The handshake procedure required to establish and validate a connection with a remote peer.
- Error handling and propagation: low-level networking errors (e.g., connection termination, transmission failures, or serialization/deserialization errors) are reported to higher layers.
- An event-driven consumer model: registered consumers can subscribe to and receive Transport events, including incoming messages and error notifications.

The most important distributed-system quality attribute addressed by the system is **eventual consistency** req. 1.

As in distributed version control systems (**DVCS**) and collaborative applications, a temporary period of inconsistency between replicas is considered acceptable in order to prioritize user autonomy and communication availability.

By design, the system assumes that the internal states of peers may diverge over time. However, provided that peers are eventually able to communicate (i.e., a synchronization or push operation is performed at some point), their states are guaranteed to converge.

This guarantee is achieved through the use of a **State Conflict-free Replicated Data Type (CvRDT)** as the underlying replication data model. The merge operation of a CvRDT is deterministic, associative, commutative, and idempotent, ensuring that all replicas converge to the same state once all updates have been propagated and merged. Further details are provided in the CRDT section 2.2.

This design choice, together with the other distributed-system decisions adopted by the architecture, influences the **message delivery semantics** used for communication within the cluster. The selected delivery guarantee is **at-most-once delivery** (see about limit. item 2).

The rationale behind this choice is as follows:

- **No retransmission mechanism is required:** a peer does not need to detect and retransmit messages that were not successfully delivered.
- **No risk of duplicate message delivery:** although duplicate suppression is not technically complex, it requires receivers to maintain a counter of previously received message identifiers for each paired peer. This introduces additional memory overhead and implementation complexity.

The adoption of at-most-once delivery is primarily motivated by two intrinsic characteristics of the system:

- **Eventual Consistency:** the loss of a message (e.g., a propagated CRDT state) may increase the duration of inconsistency between replicas, but it does not compromise the correctness of the system. As long as communication eventually occurs, replicas will converge to a common state.
- **State-Based CRDTs (CvRDTs):** synchronization is performed by exchanging complete replica states rather than individual operations. Consequently, the loss of a state update does not permanently prevent convergence, since newer states will eventually be propagated and merged by other peers. The trade-offs associated with this approach involve an ever increasing state size (see limit. item 5).

As a result, the system avoid stronger delivery guarantees such as at-least-once or exactly-once delivery, while still preserving eventual consistency.

This is also an important decision to satisfy the robustness req. 5 (i.e. losing messages will not tamper other requirements).

Another important distributed-system requirement is strong consistency with respect to the cluster server query req. 2. Every request to inspect the current cluster membership should provide a strongly consistent view of the system. The chosen approach is to introduce a single point of centralization within the cluster. While this design simplifies consistency management, it also introduces important trade-offs with significant implications for the system:

- Although fault tolerance for peers/replicas is an explicitly stated req. 3, the same guarantee does not apply to failures of the cluster server.
- If this centralized component becomes unavailable, the entire system can no longer operate correctly. This limitation is intentional but represents a critical constraint of the architecture. Possible strategies to mitigate this issue are discussed in limit. item 4.
- The cluster server is intentionally designed as a simple component that only responds to requests and does not provide proactive notifications. Consequently, whenever a replica needs to publish its state to the cluster, it must first retrieve the current cluster state from the server. Failing to perform this operation may lead to cluster partitions and prevent eventual consistency from being achieved.

For example, if two peers lose connectivity with each other, there would be no authoritative source to determine whether one of them voluntarily left the cluster or whether a communication failure occurred (a more detailed example is available in section 3.6).

At this stage, it is also important to highlight the features that are either not addressed or only partially supported by the system:

- **Security and permissions:** No authentication or authorization mechanisms are provided. Although the identifier assigned by the cluster server, together with the peer name and address, enables unique identification, these values can be easily forged. Some options about implementing authorization are discussed in item 12.
- **Memory efficiency:** Memory consumption and network traffic increase significantly as the system grows in terms of both the number of peers and the number of operations. This is particularly evident for the storage required by CRDT data structures and vector clocks (see next section 2.2).
- **Scalability:** The system is not designed for large-scale deployments. This limitation stems both from the centralized cluster server, which acts as a bottleneck, and from the communication model used to propagate updates. In particular, broadcasting updates results in a linear increase in network traffic with respect to the number of peers in the cluster.

The system can therefore be viewed from two distinct perspectives in terms of the CAP theorem:

- Communication with the **Cluster Server** prioritizes **strong consistency** over **availability**. Consequently, when the **Cluster Server** is overloaded or temporarily unavailable, operations may experience increased latency or may not be completed successfully.
- Communication between **CrdtPeers**, on the other hand, adopts a more relaxed consistency model based on **eventual consistency**. This allows temporary inconsistencies between peers, with the expectation that their states will eventually converge after successful communication and synchronization.

2.2 CRDT choices

CRDTs are a fundamental component of this system. Without them, the eventual consistency requirement (req. 1) cannot be guaranteed. Traditional DVCSs generally favor the use of CmRDTs (operation-based CRDTs) because:

- No explicit merge function is required, as the system does not operate directly on states but operations.

- New operations are treated as deltas and applied alongside other operations (no need for an ever increasing state).
- It's a particularly idiomatic approach for operation undo in versioning system

One of the most interesting examples is the application of historical data structures such as Merkle DAGs (the data structure underlying Git's history model) to CRDTs [4].

Although this approach could have been idiomatic for DVCSs+CRDT integrations, as discussed in section 2.1, it comes with important trade-offs, influences the chosen delivery semantics and in turn the final software complexity.

The final design decision was to support simpler a CmRDT algorithm while encapsulating its internal operations within a state-based representation (CvRDT). The resulting approach has the following characteristics:

- The entire state, including all operations, is transmitted at each synchronization step. Consequently, the loss of a state update does not compromise eventual consistency, even in the absence of retransmission mechanisms.
- Whenever a state is received, all previously unseen operations contained within it are merged into the local state.

This choice introduces an important trade-off: while retransmission mechanisms are no longer required, communication overhead increases as the number of operations grows.

- Although a more sophisticated delivery semantic would likely result in a more efficient system, such improvements are beyond the scope and time of the current project.

See comparison tradeoff between CvRDT and CmRDT in limit. item 1

The selected CRDT implementation was WOOT [3], a simple CmRDT originally proposed for collaborative text editing systems. The algorithm was sufficiently simple to implement within the project's time constraints while remaining capable of supporting the target use case.

One modification was introduced with respect to the original WOOT design. In the original algorithm, each WOOT operation is identified using a logical clock [1]. While this approach is adequate for collaborative editors, where convergence is the primary objective, it becomes limiting in a DVCS context and conflicts with the requirement of accurately representing causal history (req. 6).

Specifically:

- If an event happens before another event, its logical clock value will always be smaller.
- However, the converse is not necessarily true: a smaller logical clock value does not imply a causal relationship.

In other words, logical clocks may incorrectly classify some concurrent events as causally related.

To address this limitation, the original WOOT implementation was modified to use vector clocks, enabling an accurate distinction between concurrent and causally related events.

The general approach adopted for operation management is summarized below:

- Each operation is identified by a logical operation identifier composed of two elements: a site identifier, which is unique within a cluster and assigned by the Cluster Server during the cluster membership process, and a vector clock.
- Whenever an operation is created locally, the corresponding entry in the site's vector clock is incremented.
- Whenever a state is received, all received operations are used to update the local vector clock. As a result, any subsequently generated local operation will be causally dependent on the received operations.

Apart from this modification, the WOOT implementation remains identical to the one described in the original paper (see WOOT implementation details in section 4.3).

This modification also introduces a significant scalability concern. In the original WOOT design, operation identifiers are represented by a single logical value and therefore do not grow with the number of replicas. In contrast, vector clocks require one entry per replica, causing their size to increase linearly with cluster membership (limit. item 6).

Additionally, the system does not implement any garbage-collection strategy for removing vector clock entries associated with replicas that have left the cluster. This decision was made deliberately, as maintaining a complete representation of the cluster's causal history was deemed more valuable than reducing the memory overhead incurred by the growth of vector clocks.

3 Design

The system design was primarily driven by the selected **distributed systems** principles, as well as by the identified **non-functional requirements** (quality attributes) and **functional requirements**. Several concepts introduced during the requirements analysis (e.g., **communication**, **CRDTs**, and **general infrastructure**) are further refined and described in greater detail throughout the design phase.

3.1 Architecture

The overall architecture can be viewed as a **multi-layered architecture**, where each layer has a well-defined responsibility. The layers are presented starting from those closest to the user:

- **Command layer:** As introduced previously, the application operates as a CLI application. The command layer is connected to an input stream (e.g., `stdin` or command files) to parse and execute user commands. It implements a blocking command loop and provides a lightweight framework for defining handlers corresponding to the supported use cases (e.g., creating a cluster server or client, pushing data, or visualizing operation history).
- **Cluster layer:** This layer contains the application’s main entry point and implements the core logic for managing both cluster servers and cluster clients. It also provides cluster-level logging facilities.
- **CvRDT peer layer:** This layer encapsulates the responsibilities of an individual peer after it has been created by a cluster client and has joined the cluster. It implements all peer-specific functionality, such as propagating data to other peers, managing local CRDT replicas, and interpreting operation history. This layer is not used by the cluster server and is therefore relevant only to cluster clients. Both this layer and the previous one can also be considered as part of a unique **application layer** (i.e. client, server and peers provide for all of the app responsibilities)
- **Session layer:** This layer implements the delivery semantics introduced in the previous section and operates on top of the transport layer. It is responsible for automatically discarding messages under specific conditions (e.g., session timeout, out-of-order delivery or a full session queue) and for encapsulating different application-level message topics. Although its responsibilities resemble those of a Message-Oriented Middleware (MOM), it cannot be considered one because it is tightly coupled with both the application layers and the transport layer (an alternative possible approach is discussed in limit. item 14).
- **Transport layer:** This layer encapsulates low-level communication and transport-related functionality. It provides peer-to-peer communication services without making assumptions about the underlying transport protocol.

Important note: The architecture described above does not strictly follow the classical definition of a layered (or multi-layered) architecture. Instead, the layers are presented in this order to simplify the discussion of the system’s main architectural components and to clearly illustrate how they interact with one another.

3.2 Infrastructure

While peer-to-peer communication operates in a decentralized manner, allowing each peer to broadcast its current state to the other peers in the cluster, it’s important to note that all cluster management operations (e.g. cluster server discovery, server ping and peer creation) follow a client-server architecture. Accordingly, three main types of infrastructural nodes can be identified in the proposed architecture:

- **Cluster Client:** an application responsible for contacting a `ClusterServer`. Once the connection is successfully established, it creates a `CvrdtPeer` to participate in

the decentralized cluster communication. From a data locality perspective, each application can host at most one `ClusterClient` and its associated `CvrdtPeer`.

- **Cluster Server:** an application responsible for managing cluster membership. It is contacted by `ClusterClient` instances to create new `CvrdtPeer` instances or to retrieve the current cluster membership. From a deployment perspective, it may reside on the same machine as a `ClusterClient` or on a completely separate node. It's not possible to host more than a client and a server on the same machine, see limit. item 15.
- **CvrdtPeer:** a peer responsible for broadcasting state updates to the other peers within the same cluster. It is created and managed directly by its corresponding `ClusterClient`.

Each of these entities represents a replication unit within the system. A cluster can contain multiple `ClusterClient` instances and their corresponding `CvrdtPeer` instances. Furthermore, multiple clusters can coexist simultaneously, each managed by its own `ClusterServer`. As discussed in previous section, relying on a single `ClusterServer` per cluster introduces a centralization point that constitutes one of the main limitations of the current architecture.

One of the most important and reusable parts of the system is the networking and communication infrastructure. These components are shared by all application modules and provide the abstractions required for inter-node communication. They can be categorized into three main components which constitute the networking subsystem, see fig. 2:

- **De/Serializer:** provides a common interface for encoding and decoding binary messages exchanged over the network. It is assumed that communicating entities use the same concrete implementation to ensure a consistent interpretation of the exchanged messages. It is worth noting that, although message headers are encoded and decoded at the transport layer, payloads are serialized and deserialized only at the application layer. This separation is intentional: header information is required by the transport layer before message delivery (e.g., to route messages based on their metadata), whereas payload contents are application-specific and should remain opaque to the transport layer.
- **Transport:** provides a high-level API for communicating with remote peers and for receiving messages or transport-related errors. As discussed previously, no assumptions are made regarding the underlying communication protocol, allowing different transport implementations to be used interchangeably. At this level of abstraction, the component has no knowledge of the role of the remote endpoint, which may represent either a client, a server or a peer. It's important to note that despite part of its responsibilities are abstracted behind the `Session` component, some others (i.e. peer management) can be used directly by external consumers.
- **Session:** operates on top of the `Transport` layer to provide session-oriented communication and stronger delivery semantics. While the `Transport` component

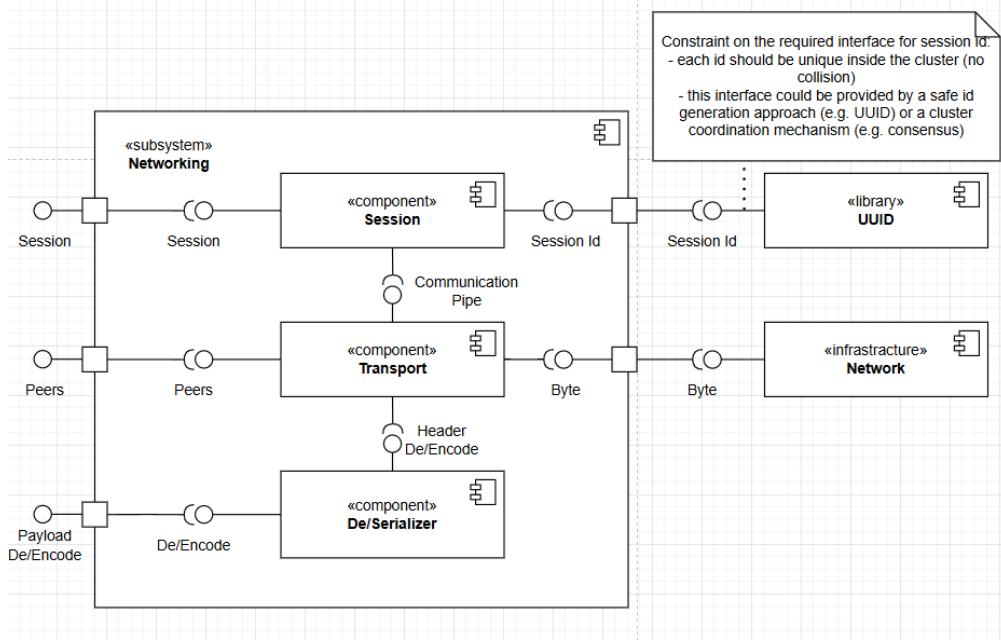


Figure 2: networking subsystem structure

exposes a simple message channel, it does not provide higher-level features such as message timeout handling, dropped-message detection, or long-lived communication sessions. One of the most challenging aspects of the **Session** component was guaranteeing that session identifiers generated by different nodes in the cluster could never conflict. A collision could cause a message to be delivered to the wrong session on the same node. Since every **Session** instance is replicated across the distributed system, coordinating the generation of globally unique identifiers would have introduced significant complexity. To avoid this issue, the component relies on UUIDs, using a library compliant with RFC 4122 [2], to generate session identifiers. Although UUIDs are considered practically unique, they still present a theoretical possibility of collision, more precise info about this problem is available in limit. item 20.

As discussed in the CRDT section 2.2, another key subsystem of the software is responsible for providing the basic data-oriented DVCS capabilities, namely the **Repository**. Its primary role is to provide the CRDT functionality required to guarantee eventual consistency within the DVCS (req. 1), while also exposing the resulting convergent data structures in a form that can be effectively consumed by external applications.

This subsystem mainly consists of three components (fig. 3):

- **CRDT**: this component implements the internal logic required to manage CRDTs. In particular, it exposes two main services to external consumers: the **CvRDT** itself, representing the complete state-based CRDT and all operations that have been applied to it, and a snapshot of its current state, providing a human-readable

representation of the underlying data structure. The exact nature of this representation depends on the specific CRDT implementation and its intended purpose; for example, it may represent a convergent numerical value or a convergent textual representation.

- **FileContext**: this component constitutes the core of the DVCS functionality. It operates on text-based CRDTs and retrieves their textual snapshots. Each **FileContext** instance is associated with a single file in the cluster, on which creation, modification, and deletion operations can be performed. Consequently, multiple **FileContext** instances may coexist within the same replica, with each instance managing a different file and its corresponding CRDT state. Another key responsibility of this component is to continuously poll representation snapshots from the associated CRDT and write them to an output stream on the node's file system. This ensures that each file in the cluster is consistently updated with the latest convergent representation of its state.
- **History**: this component provides a representation of the operations performed on a specific CRDT. Depending on the chosen representation, the history can be visualized, for example, as a linear sequence of operations or as a graph/DAG.

The purpose of this subsystem is therefore to provide both the CRDT state and its human-readable representation to external components. An important and somewhat unconventional design choice, illustrated in the figure, is that each file is associated with its own CRDT. Consequently, each file also has an independent history, with each **History** instance corresponding to a single CRDT.

In a classical DVCS, the repository history is generally agnostic to individual files: a commit represents a change to the repository as a whole and may contain modifications to multiple files. In the proposed system, by contrast, each file maintains an independent history. This represents a clear but intended limitation, introduced to better reconcile traditional DVCS functionality with the requirements of collaborative editing. In a collaborative context, modifications are typically relevant primarily to the file being edited, while changes occurring concurrently in unrelated files are generally of little interest to the users involved. Separating histories at the file level therefore simplifies the management of collaborative changes, at the cost of deviating from the repository-wide history model traditionally adopted by DVCSs (limit. item 19).

All these components provide the functionality required for a **CvRdtPeer** to operate as intended (see fig. 4):

- Each **CvRdtPeer** communicates with other peers through its own **Networking** subsystem.
- Whenever a peer receives new operations from other replicas, they are merged into its local **Repository** subsystem. Conversely, when synchronization is required, the current CRDT state is retrieved from the **Repository** subsystem and broadcast to the other peers.

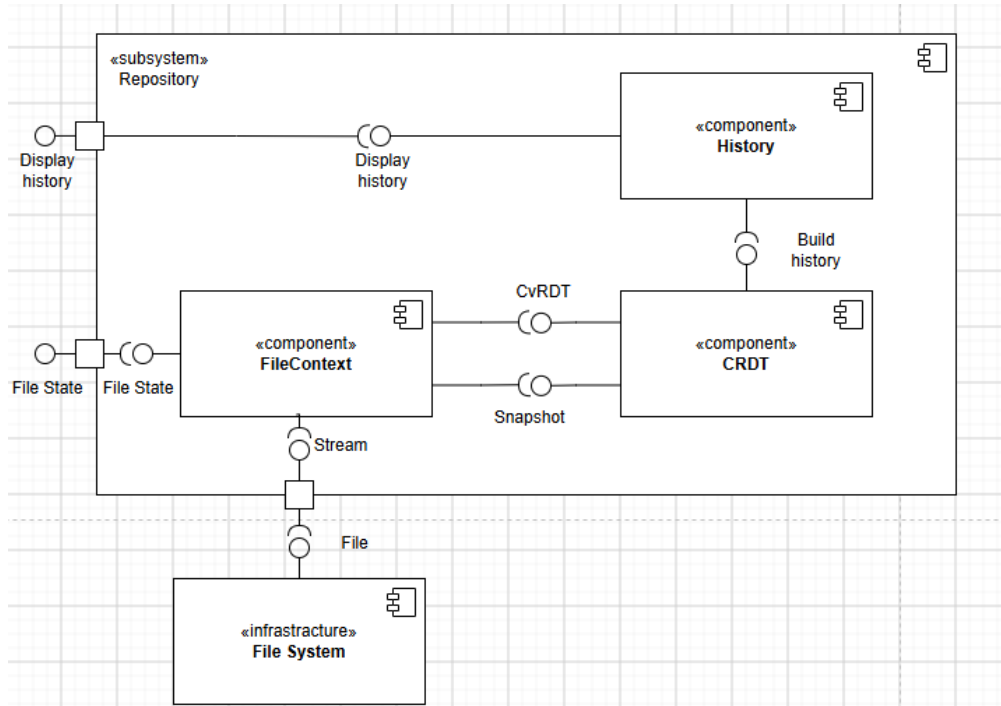


Figure 3: repository subsystem structure

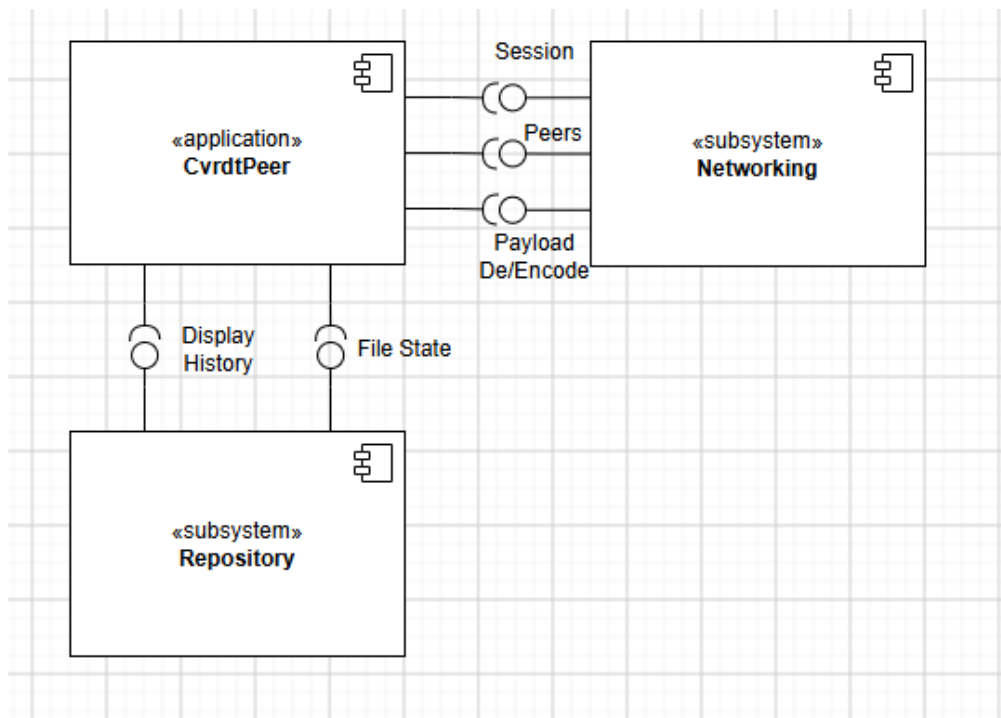


Figure 4: CvrdtPeer interaction structure

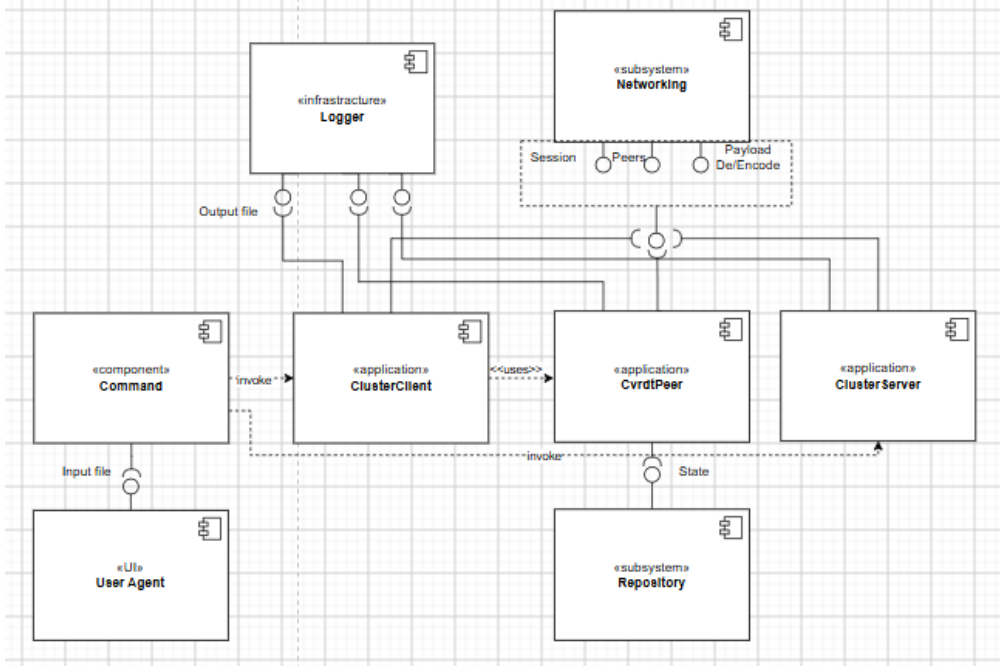


Figure 5: Full system structure

At this point, the overall software architecture can be summarized as follows in fig. 5:

- As previously discussed, all applications rely on the same **Networking** subsystem. However, only the **CvRdtPeer** depends on the **Repository** subsystem, since it is the component responsible for storing, maintaining, and synchronizing the current CRDT state.
- All applications provide an output stream to a global (singleton) logger, which records information about the execution flow and the application's internal events.
- No assumptions are made about the nature of the user agent. It may be either a human user interacting through the standard input or an automated command source (e.g., a command file). In both cases, commands are interpreted independently by both the client and the server. Consequently, a client and a server may coexist on the same machine (albeit on different ports), with each one processing only the commands relevant to its own execution.
- As shown in the architecture, the **ClusterClient** is responsible for creating, configuring, and managing the lifecycle of **CvRdtPeer** instances.

3.3 Modelling

In this section, the infrastructural components introduced in section 3.2 are described in greater detail, together with the design of the domain-specific entities.

Referring to the networking subsystem presented in fig. 2, one of the most relevant design aspects is how the transport and session services are exposed to the upper layers of the architecture. The main entities involved are illustrated in fig. 6.

- As previously discussed, the **Transport**[D, T, R] layer makes no assumptions about the underlying communication protocol (implementation details are provided in section 4.1). It can be viewed as an abstraction exposing two primary responsibilities: establishing communication channels with a set of remote targets of type T (where T is any comparable type), and providing bidirectional communication of messages of type D. The latter responsibility is delegated to the **CommunicationPipe**[D, T] sub-interface, allowing the transport and session layers to remain cleanly separated and reducing the responsibilities of the session management component.
- The transport layer is also responsible for serializing and deserializing messages exchanged over the network through a dedicated **Codec**[D]. Since the transport layer is intentionally agnostic of the concrete message type D, it relies on a message factory capable of constructing a message from a transport header of type R and a payload. This mechanism allows the transport layer to generate protocol-independent events that are delivered to the application layer, such as successful peer pairings or client-side communication errors.
- The **MessageBroker**[D, T] supervises the exchange of messages through a dedicated **CommunicationPipe**[D, T] with one or more remote targets of type T. It represents the main entry point to the session services described in fig. 2. The broker exposes a higher-level API by allowing the registration of **TopicHandlerFunc**[T, D] handlers for specific **Topics**. Whenever a message is sent or received on a given topic, a dedicated **Session**[D, T] is created to manage the corresponding communication.

The entities described so far are intentionally generic, as they are designed to be independent of the underlying transport protocol, serialization strategy, and message broker implementation. Nevertheless, it is equally important to describe the type of data exchanged through the networking subsystem. In the current design idea, the transport and session layers operate on messages of type **PeerMessage**, whose structure is illustrated below in fig. 7:

- A **PeerMessage** consists of a header (**PeerMetadata**) and a **Payload**. While the header has a fixed structure, the content of the **Payload** varies according to the specific message being exchanged (see section 3.5 section for a deeper explanation of all the topic events).
- The **PeerMetadata** contains information identifying the sender. In particular, the **PeerHostInfo** field stores the unique identifier of the peer using the format `<id>_<username>_<address>` (this specific identifier is also used by **Cluster Clients** and **Cluster Servers**, with them being assigned a special id for clients and servers).

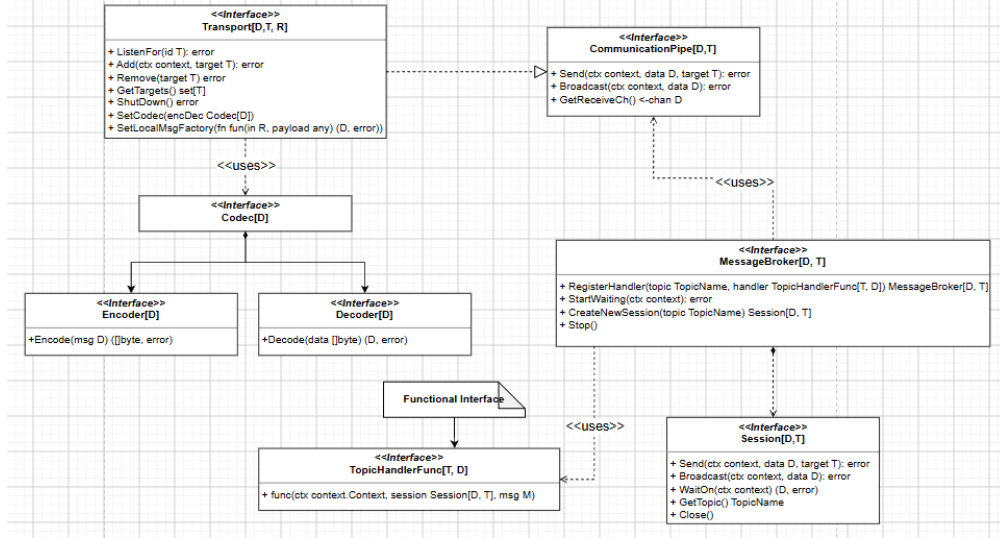


Figure 6: Transport and Session generic structure

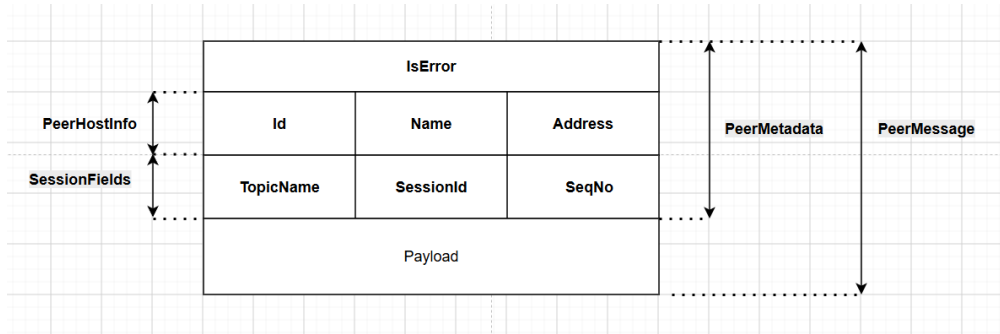


Figure 7: PeerMessage data packet

- The **SessionFields** structure contains all the information required to uniquely identify the destination **Topic**, the associated **Session**, and the message **SequenceNumber**, allowing the session layer to correctly route and correlate messages.

As discussed previously, the transport and session layers are only responsible for encoding and decoding the header portion of a message.

The underlying idea is that only the application-level receiver should possess sufficient knowledge to interpret the message payload, whereas the lower layers should interact with it solely through its metadata. To support this separation of concerns, the architecture introduces the following abstraction fig. 8:

- An **Envelope[K]** is composed of a comparable key of type **K** and a **Payload**.
- While the key is directly interpretable by the transport and session layers, the **Payload** exposes only a decoding interface, allowing the application-level receiver to determine and reconstruct its concrete type.

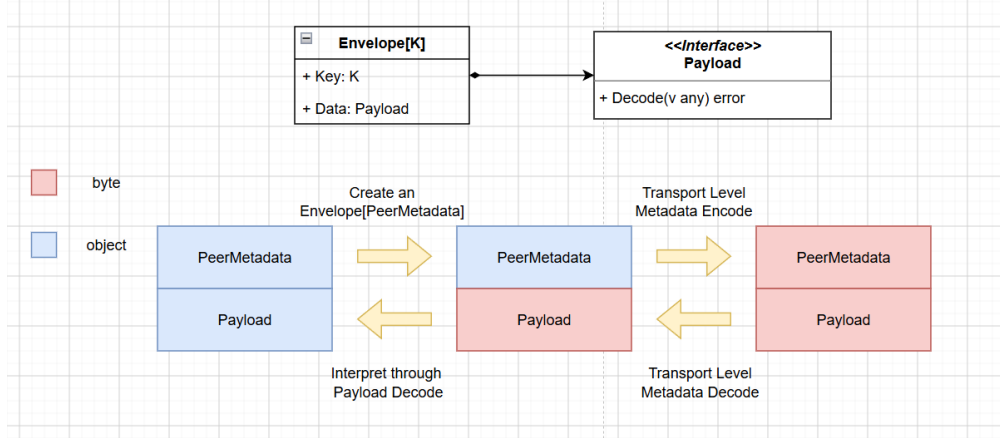


Figure 8: Envelope abstraction working

- In other words, an **Envelope** provides a partial encoding/decoding abstraction that cleanly separates transport-level metadata from application-level data.
- In the current implementation, the **PeerMessage** described previously is a concrete instance of this abstraction and can be represented as an **Envelope[PeerMetadata]**.

This abstraction also enables the construction of hierarchical message formats, where multiple **Envelope** instances can be nested within one another. Each layer may employ a different **Codec**, allowing independent serialization strategies to be applied to different portions of the message while preserving the separation of concerns.

At this point, three main responsibilities can be identified within the network subsystem:

- **Transport responsibilities**
- **Session responsibilities**
- **Application-level de/encoding**

This is mapped to a **NetworkModule** from the point of view of design entity, every element that needs to actively interface and communicate over the system needs to be injected with it.

The **NetworkModule** does not introduce additional behavior on its own; rather, it serves as a unified abstraction that groups the three aforementioned responsibilities into a single structure. This simplifies dependency management and provides a consistent interface for components that require network communication.

To summarize, every **ClusterClient**, **ClusterServer**, and **CvrdtPeer** needs to own a **NetworkModule** in order to provide its basic functionality.

However, two additional pieces of information are required for these components to operate as fully-fledged distributed nodes:

- **A clear identifier:** each node in the cluster, whether it is a peer, client, or server, needs a unique identifier that can be used for both local and global identification.
- **A logging utility:** each node needs its own independent logger to record and manage events occurring within the node.

The final entity responsible for providing the functionality required by a node to operate properly is the `AppNodeModule`. The overall structure and responsibilities of this module are illustrated in the following fig. 9.

Some important aspects of its design should be highlighted:

- Each `AppNodeModule`, in addition to a node identifier and the previously discussed `NetworkModule`, is provided with a `GlobalLogger`. This interface defines a simple and consistent contract for implementing a system-wide logging facility.
- Some components of the system may require pluggable configuration. This is supported through the `Configurable` interface, parameterized by a specific `SupportedConfiguration`. In the current implementation, the supported configurations are `TransportConfig`, `BrokerConfig`, and `LoggerConfig`.
- The `Envelope` is an additional component responsible for providing application-level message encapsulation. It allows payloads to be wrapped into an `Envelope`, providing a common structure for messages exchanged at the application level.

The adoption of generics may initially appear somewhat rigid; however, it provides a useful separation of concerns between the underlying infrastructure and the data flowing through it.

As previously discussed, the `PeerMessage` data packet in fig. 7 is the data structure adopted as the standard message format throughout the application. This structure is also used to instantiate the generic `AppNodeModule[D, T, R]` as follows:

- **D:** represents the generic type used to identify the data flowing through each node. In the adopted implementation, this corresponds to the complete packet, namely `PeerMessage`.
- **T:** represents the type used to identify the target entity of each node. In the adopted implementation, this corresponds to the header identifier contained within the message, namely `PeerHostInfo`.
- **R:** represents the type of the metadata associated with the message header. In the adopted implementation, this corresponds to `PeerMetadata`, which is used to provide the additional information required by the message envelope.

The resulting specialization of `AppNodeModule[D, T, R]` is therefore `AppNodeModule[PeerMessage, PeerHostInfo, PeerMetadata]`. To simplify its usage and notation, this specialization is aliased as `PeerAppNodeModule`, explicitly indicating its dependency on the `PeerMessage` data structure.

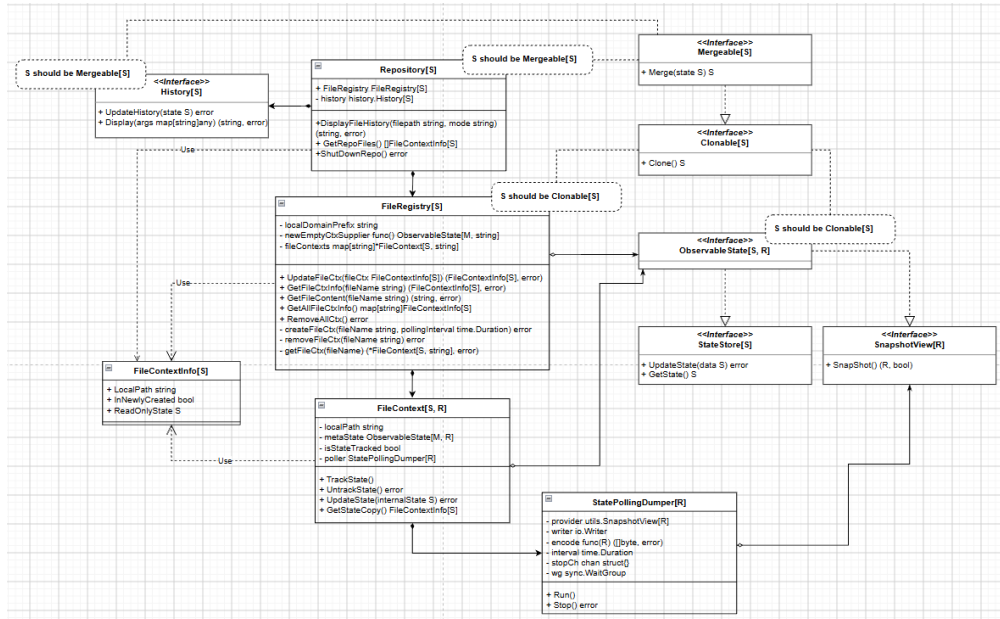


Figure 10: Repository structure

fensive copies of the state when exposing information about a file. In particular, it allows the registry to return a read-only representation of a context through `FileContextInfo[S]`, preventing external components from directly modifying the state maintained by the repository.

As can be seen from the generic specification of the implementation, no assumptions are made about the concrete type of state managed by the `FileRegistry`. Instead, only two requirements are enforced:

- The underlying state S must support state-merging operations by implementing the `Mergeable[S]` interface.
- A textual representation of the state must be provided for snapshots.

These requirements can be satisfied by the `ObservableState[S, string]` interface, which combines access to the underlying state with the ability to provide also a textual snapshot of its current representation. This abstraction can, for example, be used to implement a text-based CRDT as follows:

- A text-based CRDT maintains an internal convergent state S , which must support state-merging operations.
- The same CRDT can provide a human-readable snapshot of the underlying state, such as the resulting text of a file. Although requiring the CRDT itself to provide this representation is not strictly necessary, it is useful because it separates the CRDT's internal data structure and convergence logic from the resulting representation exposed to external consumers.

This design therefore allows the `FileRegistry` to remain agnostic of the concrete CRDT implementation while enforcing only the properties required by the storage subsystem: the ability to merge states and the ability to expose their current representation as text.

The generic type adopted for the underlying state S is `CvRDTState`, illustrated in fig. 11. As introduced in section 2.2, the design of the `CvRDT` models its state as the set of all operations that have been received and are currently stored. Consequently, merging a new `CRDTOperation` into a `CvRDTState` consists of adding the operation only if it is not already present in the state.

Each operation can represent either an `Insertion` or a `Deletion`, corresponding, for example, to the insertion or deletion of a character within a file. Every `CRDTOperation` also contains a `LogicalId`, which identifies the original site that generated the operation and, more importantly, stores the vector-clock state associated with the operation at the moment of its creation. This information is essential for determining the causal relationship between operations.

The `Repository` therefore instantiates its generic abstractions using `CvRDTState` as the underlying state type. The same state type is used both by the `History` component and by the `FileRegistry`, ensuring that the operation state maintained by the repository is consistent across its different components.

It is important to note, however, that `CvRDTState` represents only the state model exposed to the repository and does not constitute the complete CRDT implementation. The concrete CRDT remains an implementation detail of the repository. In particular, the actual CRDT can be implemented as an `ObservableState[CvRDTState, string]` and injected into the repository. This component is responsible for providing the actual CRDT algorithmic logic, including state management, merging, and the generation of the textual snapshot, while the repository remains concerned with storing and exposing the resulting state (see section 4.3 for CRDT implementation details).

At this point, the application-layer entities that rely on these concepts can be described more precisely:

- A `ClusterNodeStore` is the main application entity. It stores the current local `Cluster Client` and `Cluster Server` (see limit. item 15) and supervises their initialization and shutdown.
- Both the client and the server implement a `Node` interface, which provides a set of common methods for managing them.
- As previously described, both the client and the server own an `AppNodeModule`, which provides the services required by the corresponding node. The module is injected into each newly created node by the `ClusterNodeStore` through a dedicated factory.
- For the `CvrdtPeer`, the `Repository` was extended into a `CvrdtRepository` to provide CRDT-specific functionality, such as high-level operation creation and the interpretation of repository state received from other peers. This functionality is implemented through a CRDT-implementation-dependent `adapter`.

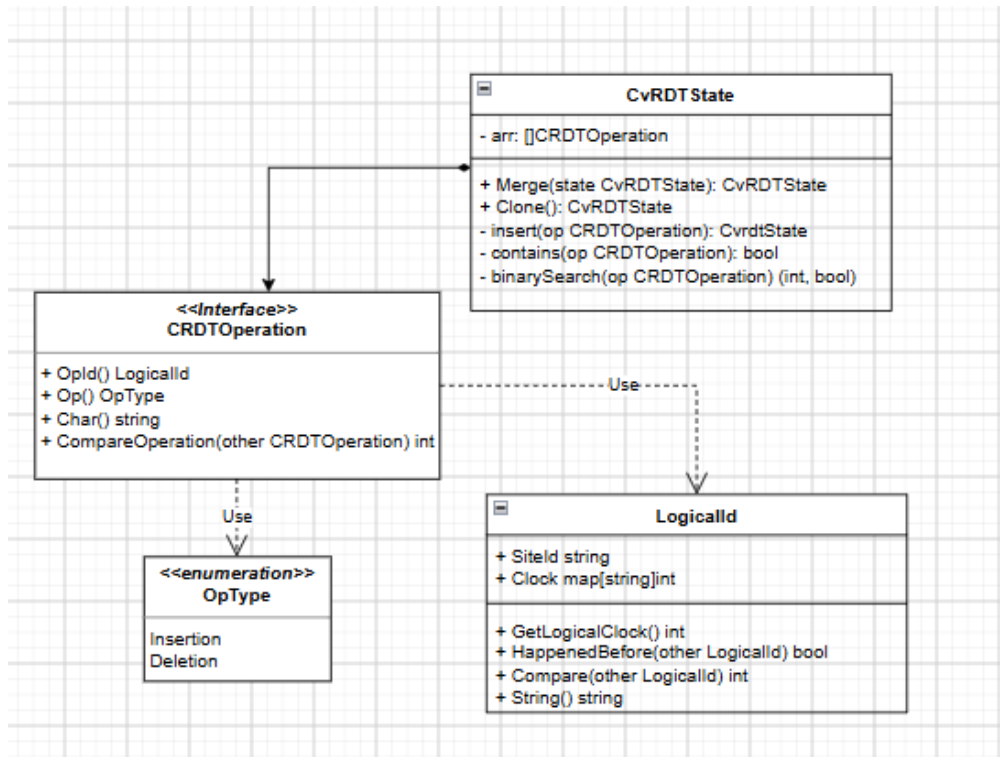


Figure 11: CvRDTState structure

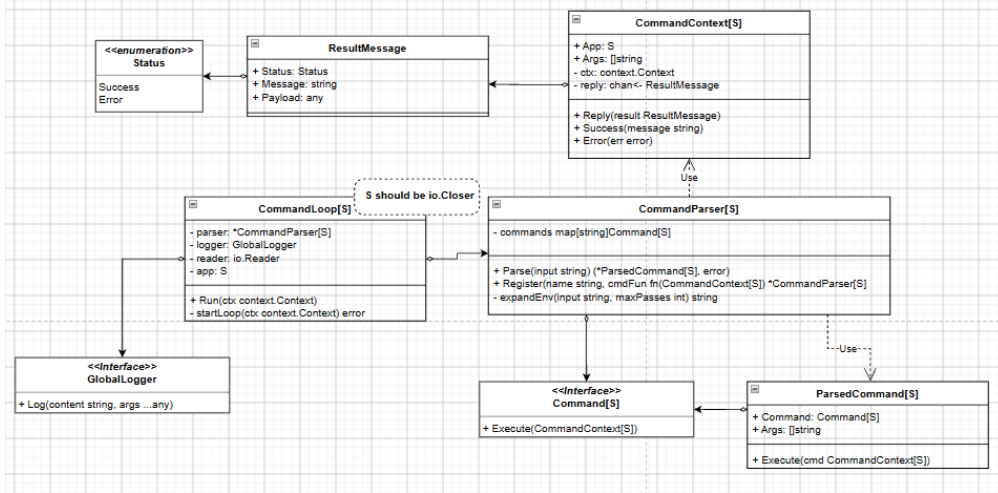


Figure 12: Command class diagram structure

From the user’s perspective, the application layer is never accessed directly. Instead, users interact with the system through a command component, whose entities abstract the underlying application layer; its entry point is the `ClusterNodeStore`. The entire software is executed through a command loop, which provides the following features (see fig. 12):

- A `CommandLoop[S]` is a blocking, single-threaded loop that allows users or other input sources (e.g., command files) to execute commands and interact with the application layer through a generic `io.Reader`.
- The generic type S makes the module reusable with different application implementations. In the actual system, it is instantiated with a `ClusterNodeStore`. The type S is required to implement `io.Closer`, providing a mechanism to shut down the application before terminating the command loop.
- Each line received from the input stream is interpreted as a command by a `CommandParser`, to which command handlers can be registered. The parser also supports environment variable expansion, which is particularly useful for creating configurable integration tests (see section 6.2.2).
- Each command handler is provided with a dedicated `CommandContext[S]`, through which it receives a reference to the application instance S , together with a Go `context.Context`, the command arguments, and a result channel through which it can report its execution result.
- The result of each command, whether successful or unsuccessful, is logged through a `GlobalLogger` object.

3.4 Interaction

As discussed previously, the application entry point is a command-line interface (CLI). More specifically, it's a blocking CLI application, meaning that every command issued by the user agent terminates only under one of the following conditions: the operation completes successfully, it fails, or a timeout expires.

Conversely, replica components such as `ClusterClient`, `ClusterServer`, and `CvrdtPeer` operate asynchronously. Each component maintains its own control flow to continuously listen for incoming messages over the transport layer and on the appropriate session topics.

The first key interaction in the software is the cluster membership procedure. For simplicity, the following example assumes that the `ClusterServer` is created by the same user agent that subsequently creates the corresponding `ClusterClient` (see fig. 13):

- The first step consists of creating a `ClusterServer` (or connecting to an existing one). Once the server is available, a `ClusterClient` can be created. Each client is uniquely identified by a username and a network address. At this stage, no network connection is established; instead, the server's connection information is stored within the client for future use, resulting in a one-to-one association between a `ClusterClient` instance and a `ClusterServer`.
- After its creation, the `ClusterClient` can initiate the cluster join procedure by contacting the server. This operation begins with a transport-layer handshake. If the handshake does not complete before the configured timeout expires, the connection attempt is aborted and the operation returns a failure (see section 4.1 for handshake details).
- If the handshake succeeds, the `ClusterClient` opens a new session and sends a join request on the `ping_join` topic. The request includes the client's information (username and network address) together with the address of the `CvrdtPeer` that the client intends to register.
- The `ClusterServer` validates the request and returns an error through the established session if the client information are already associated with another cluster member, if the request payload is invalid, or if a session timeout occurs.
- If the validation succeeds, the `ClusterServer` assigns a new monotonically increasing cluster identifier to the client and returns it together with the list of existing `CvrdtPeer` instances in the cluster. Each peer is identified by a tuple of the form `<id>_<user>_<address>`.

The cluster membership join procedure illustrated in fig. 13 omits an important step that is required to fully synchronize a newly created `CvrdtPeer` with the cluster. Before the join operation can return either a success or a failure, the `CvrdtPeer` must align its current set of remote peers with the membership information received from the `ClusterServer`. This alignment procedure is illustrated in fig. 14:

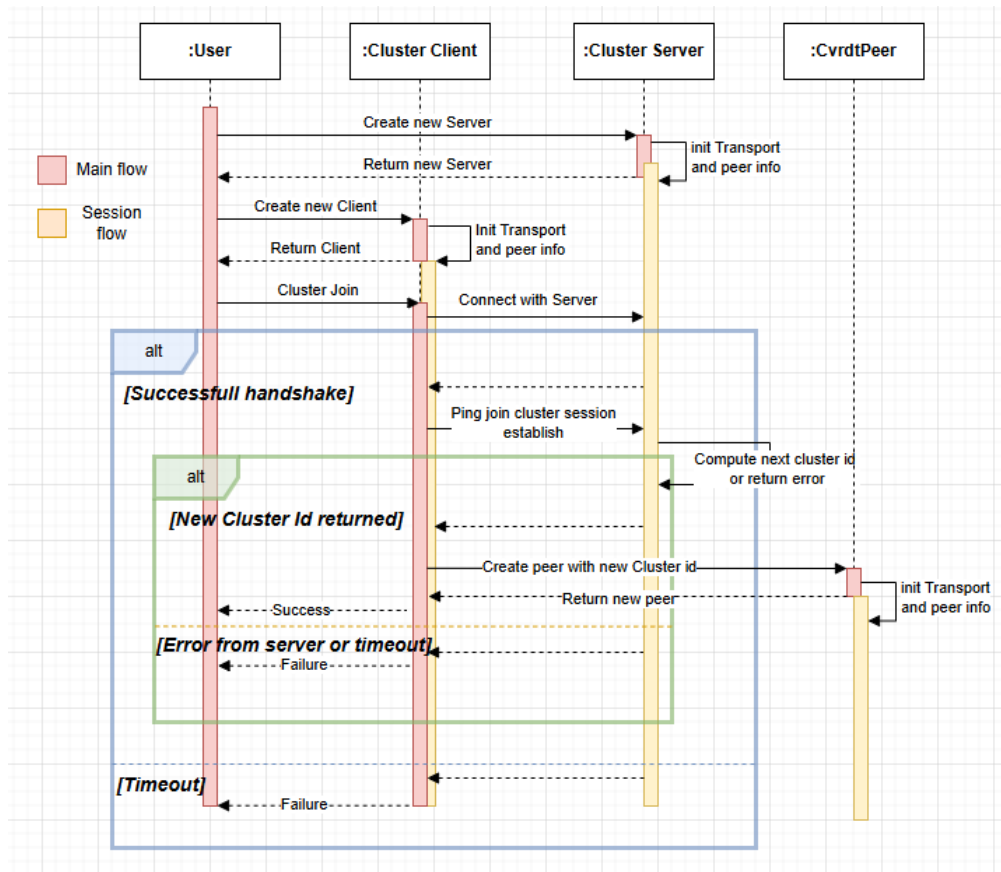


Figure 13: Cluster membership join interaction

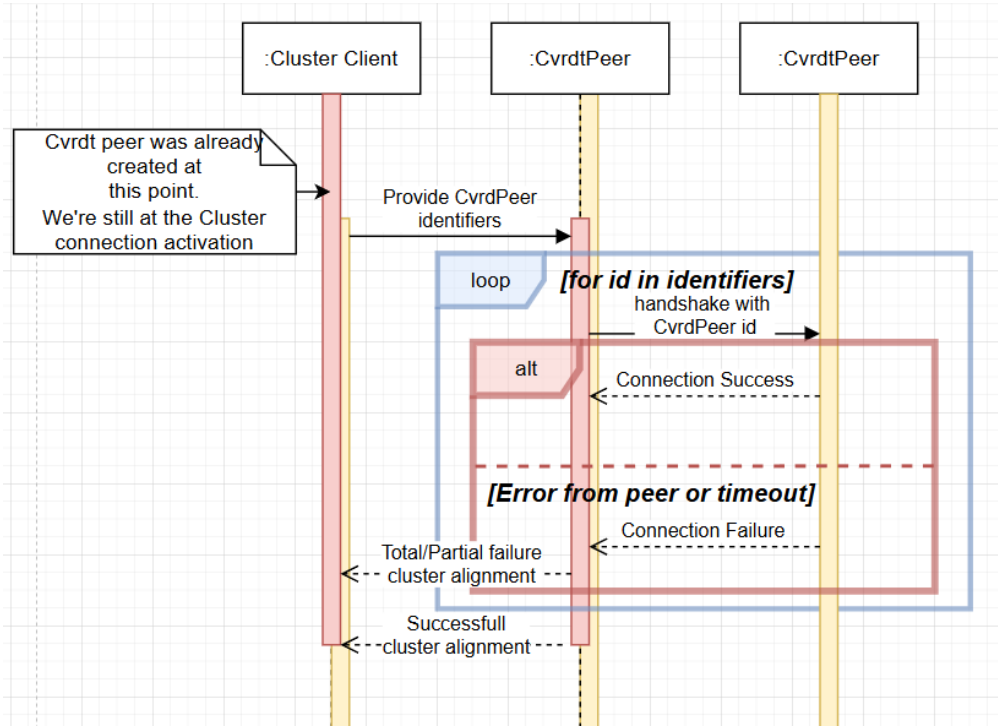


Figure 14: CvrdtPeer alignment interaction

- Using the identifiers returned by the `ClusterServer`, the `CvrdtPeer` establishes connections with the other peers currently registered in the cluster.
- As shown in the figure, this procedure may result in a partially synchronized state. If the connection attempt to one of the peers fails, the overall operation returns a failure even though some peer connections may have already been successfully established.

It is important to note that a partial failure during this procedure does not compromise the distributed properties of the architecture. Instead, it only results in a temporary weaker degree of synchronization between the newly joined `CvrdtPeer` and the rest of the cluster. This temporary inconsistency is naturally resolved through subsequent communications among the peers.

For this reason, it is also important to describe the interactions that occur during the normal communication phase of the cluster.

fig. 15 illustrates the procedure through which a `CvrdtPeer` propagates its current state to the other peers in the cluster. Using terminology borrowed from distributed version control systems (DVCSs), this procedure can be regarded as a *broadcast push*. The process can be divided into three main operations:

- **ClusterServer ping retrieval:** The first step consists of retrieving the current list of registered peers from the `ClusterServer`. This operation is essential to

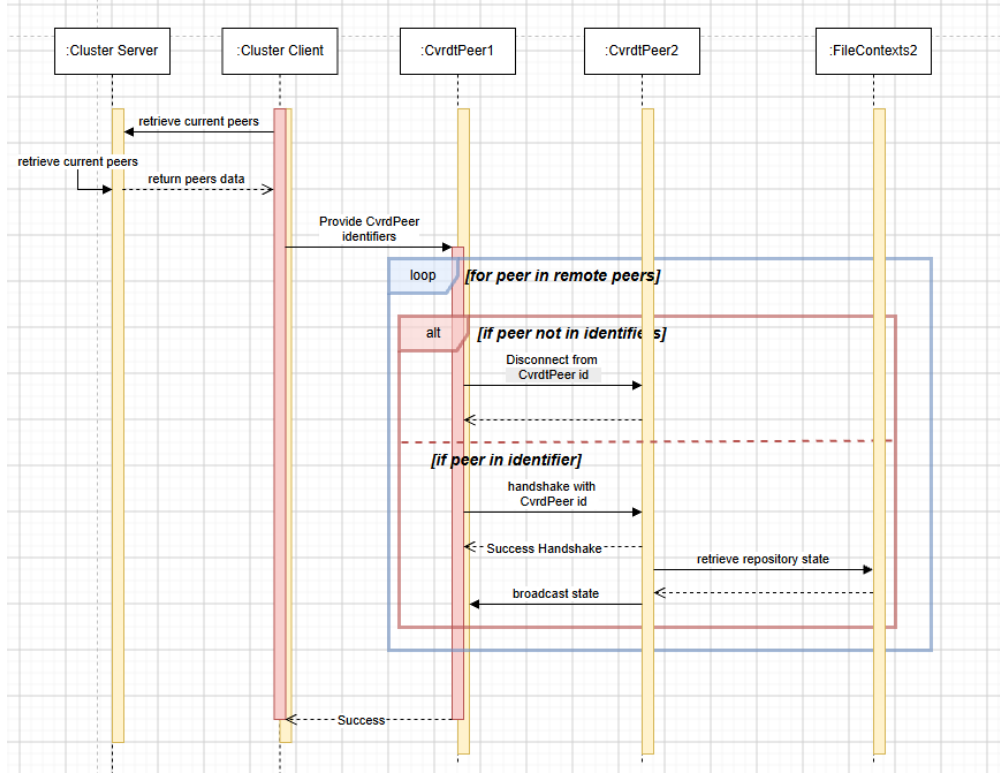


Figure 15: CvrdtPeer broadcast push interaction

improve the consistency of the local membership view, as new peers may have joined the cluster but previous connection attempts may have failed (see more about consistency analysis in section 3.6).

- **Peer alignment:** As in the cluster join procedure, the **CvrdtPeer** establishes connections with newly discovered peers and disconnects from peers that are no longer registered in the cluster.
- **Broadcasting:** the newly paired **CvrdtPeer** will broadcasts all of its repository file contexts (i.e., the CRDT state associated with each file) to the previously connected peers and the one which started the connection. Upon receiving a context, each remote peer merges the received CRDT state into its local store. It's important to note that this broadcasted state will be received asynchronously by every peer and therefore will not interfere with the success of connection procedures.

3.5 Behaviour

This section describes how each of the three infrastructural entities reacts to events and messages:

- **Cluster Server**
- **Cluster Client**
- **CvrdtPeer**

It is important to note that, although the **CvrdtPeer** is locally invoked by the **Cluster Client**, as they reside on the same machine but listen on different ports, the two components have independent **AppNodeModule** instances. Consequently, they operate as independent network nodes and react independently to incoming network messages.

The **Cluster Server** is the only infrastructural component whose behavior is exclusively driven by incoming events. Specifically, it reacts to three types of messages, each associated with a dedicated session topic. Access to the server state is mutually exclusive, ensuring that state modifications are performed sequentially and providing the strong consistency guarantees required for cluster membership management. The three events are:

- **PING JOIN**: requests the registration of a new peer in the cluster. The payload contains the address that will be used to host the new peer, which is required by the server to register it. The server constructs a new **PeerHostInfo** using the information provided in the sender's **PeerMetadata**, replacing the custom client identifier with a newly assigned auto-incremented identifier for the new **CvrdtPeer**. The server also returns the information of all currently registered **CvrdtPeers**. This event modifies the server state by adding the new peer information and incrementing the counter used to assign identifiers to subsequent peers.
- **PING RETRIEVE**: requests the current cluster membership information. Unlike **PING JOIN**, this event does not modify the server state and only returns the information of the currently registered peers.
- **PEER EXIT**: notifies the server that a remote peer has disconnected. From the server perspective only the **Cluster Client** can generate this event when disconnecting from the server. The server consequently removes the corresponding peer information from its membership state.

The **Cluster Client** is the main component through which the user interacts with the cluster. Its primary responsibilities are to communicate with the **Cluster Server** and create the local **CvrdtPeer**. It reacts to two main event topics:

- **PEER_CONNECTION**: this event is received when a connection with a remote node is successfully established. For the **Cluster Client**, this corresponds to the connection with the **Cluster Server**, as the client is the only one that can establish server connections through the transport layer. The event may also contain an error when an incoming handshake fails.

- **PEER_EXIT**: this event is received when the **Cluster Server** unexpectedly disconnects from the client. The client can consequently handle the loss of its connection to the cluster server.

The **CvrdtPeer** is responsible for managing the local **Repository** and broadcasting its state to the other **CvrdtPeers** in the cluster. It reacts to the following events:

- **PEER_EXIT**: this event is received when another **CvrdtPeer** leaves the cluster. No specific state-management action is required in this case; the event is simply logged.
- **PEER_CONNECTION**: when the event does not contain an error, it indicates that a connection with another **CvrdtPeer** has been successfully established. The local peer responds by broadcasting its current repository history to the newly connected peer. This mechanism reduces the time window during which the newly joined peer may remain inconsistent with the rest of the cluster.
- **PUSH**: this event contains the current repository state of a remote **CvrdtPeer**. Upon receiving it, the local peer merges the received state into its local repository, allowing the two replicas to converge.

3.6 Data Consistency and Availability

The system primarily stores data concerning the files defined under a specific **Cluster Server** domain. In particular, the content of each file is written to the local path of the **CvrdtPeer**, under the corresponding domain folder.

Two important limitations should be highlighted regarding the persistence and reliability of these files:

- While the resulting state is periodically written to its associated file, no information about the CRDT operations that generated the state is persisted.
- The file content is periodically polled by the **CvrdtPeer** from the source CRDT. Therefore, temporary inconsistencies between the CRDT state and the file content may occur between two consecutive polling operations.

Despite these limitations, assuming communication between replicas is eventually executed and sufficient time is allowed for all operations to propagate, the state of each file managed by **CvrdtPeer** instances belonging to the same **Cluster Server** domain is guaranteed to converge to the same content.

This consideration forms the core of the **Convergent Testing** test system (section 5.2.1) and provides an important validation of the **eventual consistency** req. 1.

Each **CvrdtPeer** shares its repository information with the other peers in the cluster. More specifically, the information shared consists of the list of all **FileContextInfo** instances, representing the operations and metadata associated with each file for which the current peer has at least one defined operation. The message format used to exchange this information is illustrated in fig. 16.

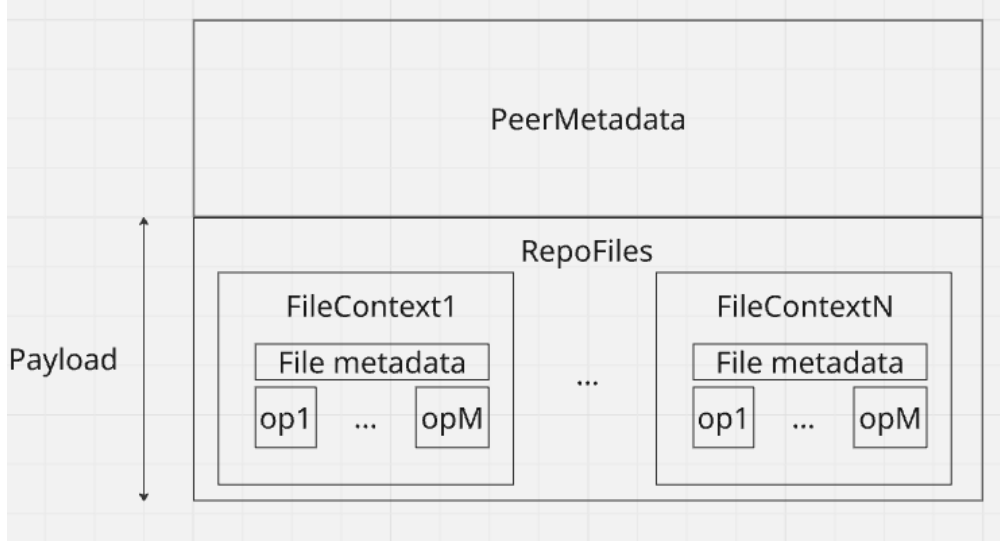


Figure 16: Data broadcast and shared with other peers

As previously discussed, sharing this type of information introduces some important trade-offs, most notably the ever-growing state problem described in limit. item 5. However, this approach also eliminates the need to implement a dedicated retransmission protocol, as missing operations can be recovered when peers exchange their repository state.

The last important piece of information needed to understand how the system currently operates concerns what is effectively stored by the **Cluster Server**. As described in the **Behavior** section 3.5, the server is a reactive component that maintains its internal store consistently across concurrent requests.

This store, namely **ClusterStore**, maintains information about the **CvrdtPeers** currently joined to the cluster, as well as the current auto-incrementing identifier that will be assigned to new peers.

It is important to note, however, that while the information provided by the store is consistent across multiple requests, since the **ClusterStore** is protected by a lock, it does not necessarily always reflect the actual state of the cluster. In some cases, this discrepancy may prevent a client from correctly synchronizing with the cluster, despite there being no actual error in the system.

For example:

- The server currently has no **CvrdtPeers** participating in the cluster.
- A client sends the server a request to join the cluster and be assigned an identifier for a new peer. At this point, the **ClusterStore** is locked to prevent consistency issues.
- While the request is being processed, another peer attempts to join the cluster.

However, the store cannot be updated until the previous request has been completed.

- The first client retrieves information from the server indicating that no other peers are currently participating in the cluster. It then creates a `CvrdtPeer` which, according to its view, is the only peer in the cluster.
- The second peer is subsequently admitted to the cluster and is provided with information about the first peer.
- Suppose that the second peer is unable to contact the first peer. In the absence of a retransmission mechanism, it must assume that the first peer has effectively left the cluster, as there is no authoritative source available to confirm its actual status.
- As a result, both peers are registered as members of the same cluster, but they have no communication path between them, resulting in a cluster partition.

To prevent this issue, a ping mechanism was adopted. Whenever a peer needs to broadcast its data, it first pings the server to retrieve the current list of peers and, if necessary, establish connections with newly discovered peers. Although this approach may increase the window of potential inconsistency caused by a cluster partition, it provides the assurance that, assuming communication is eventually restored, the system will eventually become aligned again.

This solution has some drawbacks, and more sophisticated approaches could have been adopted, as discussed in limit. item 7).

4 Implementation

In this section, further details concerning important implementation aspects of the system are presented.

This section also complements the details described in the section 3, which could not be fully specified at the design level due to their dependence on the specific implementation.

These details also provide insight into how the initial design directives were effectively implemented in the final system.

4.1 Transport TCP Implementation

In section 3, the **Transport** abstraction was introduced with the purpose of decoupling the higher-level components from the following aspects of network communication:

- the adopted transport protocol;
- header-level serialization and deserialization;
- low-level peer management procedures.

The concrete implementation adopted by the system is based on TCP, with the **Transport** interface implemented by the **TCPLayer** component.

Each new connection with a remote peer is managed through a transport-level handshake. The **Transport** abstraction does not impose any specific handshake algorithm; however, the current implementation provides a **TCPHostExchanger** to perform this procedure. Its purpose is to ensure that, before the connection is fully established, both endpoints exchange the information required to identify the remote peer, namely their respective **PeerHostInfo**.

The handshake procedure is illustrated in fig. 17 and consists of the following steps:

- **TCPACK**: the final acknowledgment of the TCP connection establishment, after which the TCP connection is established;
- **REC-EXCH**: the receiver sends its **PeerHostInfo** to the initiator;
- **SEN-EXCH**: the initiator sends its **PeerHostInfo** to the receiver;
- **ACK**: the receiver sends a final acknowledgment, completing the host information exchange.

The handshake can fail under three possible circumstances:

- the handshake procedure times out for either participant. The timeout is defined by the **TransportConfig** through **InPairTimeout** and **OutPairTimeout**;
- a message is received out of sequence. This is detected using the **SeqNo** field of **PeerMetadata**;
- either participant shuts down unexpectedly.

In all these cases, the first participant to detect the error promptly closes the TCP connection.

Another important responsibility of the **TCPLayer** is the management of **TCPPeer** entities.

While the **TCPLayer** provides support for higher-level communication aspects, such as serialization and deserialization, peer management, and handshake procedures, a **TCPPeer** is responsible for low-level communication with a remote host.

A particular problem previously highlighted is the increasing size of a **Repository** in terms of the number of operations. Since the whole state is broadcast each time, the size of the transmitted data grows as the repository grows. This behavior is intentional and is discussed in limit. item 5. However, it also prevents the use of a fixed-size buffer for reading from the TCP stream, as a message may exceed the buffer capacity, potentially resulting in an incomplete packet or incorrect interpretation of subsequent data.

For this reason, a **ByteStreamProcessor** component is provided within the **TCPPeer**. Its responsibilities are the following:

- Before each serialized byte packet, a header containing its byte length is prepended. In the current implementation, this value is encoded in big-endian format.

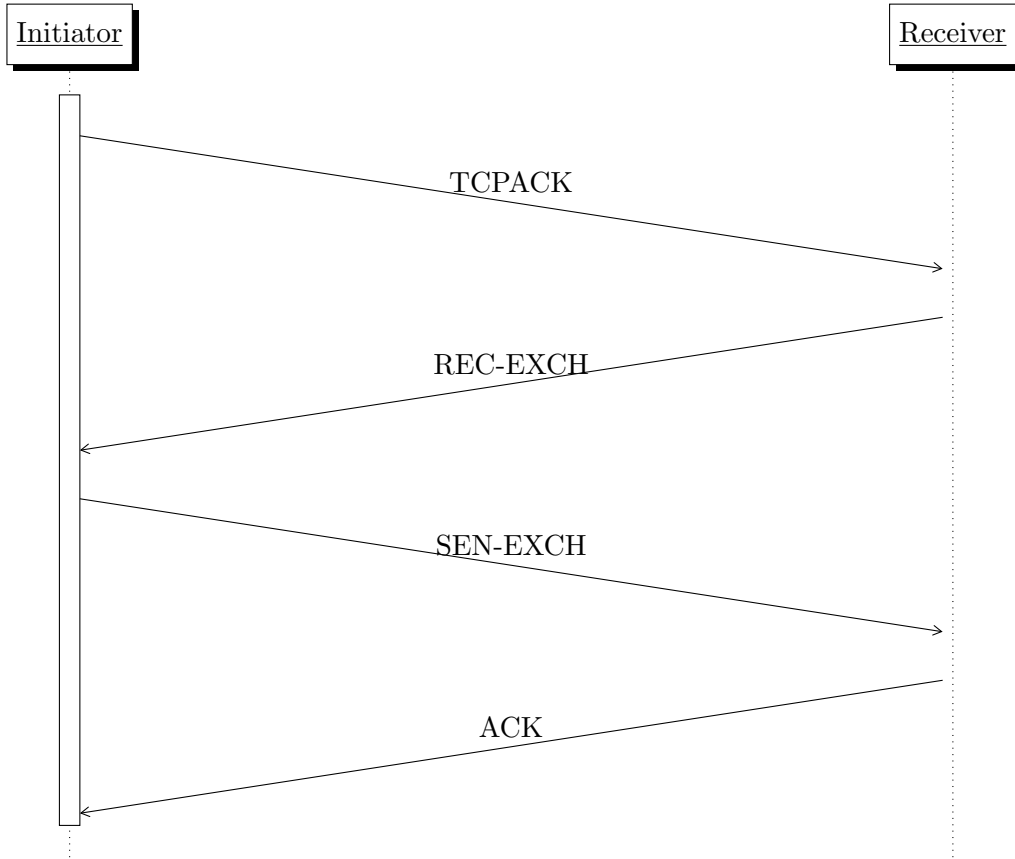


Figure 17: **TCPHostExchanger** handshake protocol

- A **MAX_MESSAGE_SIZE** constraint is enforced to prevent a remote peer from declaring an arbitrarily large packet and causing unbounded resource consumption. The current limit is set to 4,MB.

The latter constraint introduces an important technical limitation to the system: as the cluster grows in terms of the number of operations, the size of the state eventually exceeds the maximum message size, at which point the system can no longer exchange the complete repository state. The implications and possible interpretations of this limitation are discussed in limit. item 18.

4.2 Json De/serialization

The chosen de/serialization approach is the JSON data format. While JSON makes encoding and decoding relatively straightforward, it has some drawbacks compared to more compact binary formats (see the discussion on JSON de/serialization in limit. item 16).

As introduced in the design modelling in section 3.3, de/serialization is handled in two steps through the **Envelope** abstraction. This approach was implemented through the

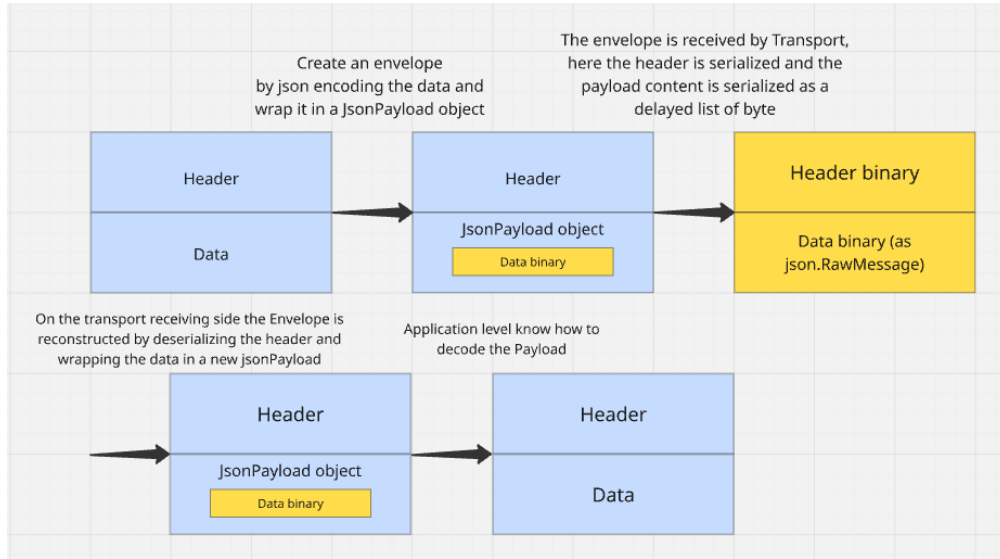


Figure 18: How envelope de/serialization works with json

JsonEnvelope class. Its internal operation is described in fig. 18 and can be summarized as follows:

- At the application level, when encoding a message, the payload (if present) is serialized using JSON marshalling, producing a **JsonPayload** object that implements the **Payload** interface.
- A new envelope is then created by associating the message header (i.e., **PeerMetadata**) with the previously created payload object.
- At the transport level, the complete message is serialized into JSON. The payload is represented as a **json.RawMessage**, a special Go type that allows part of the JSON data to remain serialized and be decoded at a later stage. If no payload is present, the corresponding JSON field is empty.
- On the receiving side, the transport layer deserializes the JSON message and extracts the header. The payload remains serialized because it is represented by **json.RawMessage**.
- The application layer then receives the message and invokes **Payload.Decode** to deserialize the internal payload data.

4.3 WOOT implementation

WOOT (*WithOut Operational Transformation*) is a conflict-free replicated data type (CRDT) specialized for collaborative editing of sequences, such as text documents. Its name derives from its opposition to Operational Transformation (OT), where consistency

of the resulting sequence is instead coordinated by a central authority. By relying on a decentralized conflict-resolution mechanism, WOOT is particularly well suited to peer-to-peer (P2P) systems.

WOOT represents a document as a sequence of uniquely identified elements, referred to as **W-Characters**. It defines two types of operations in terms of them:

- An insertion is modelled as $ins(a \prec e \prec b)$, which inserts element **e** between elements **a** and **b**.
- A deletion is modelled as $del(e)$, which makes element **e** invisible. This is typically implemented using a tombstone mechanism, allowing the element's identifier to remain known to other peers without displaying the element in the document.

The core of the WOOT algorithm, where operations are internally processed and merged, is called a **Site**. Each **CvrdtPeer** instantiates its own **Site**, using its assigned cluster identifier as the corresponding site identifier.

Because operations refer to stable element identifiers rather than mutable positions, they can be applied in different orders on different peers while still converging to the same final document state.

Here, $\prec$ represents a binary relation over the set of W-Characters that defines their ordering in the sequence.

A W-Character is defined as shown in eq. (1) and consists of the following fields:

- **id**: a unique identifier represented by the tuple $\langle \text{site-id}, \text{vector-clock} \rangle$. The original WOOT algorithm uses a simple logical clock; a vector clock was adopted in this implementation to address some of its limitations (see limit. item 6).
- α : the character content. It can represent either a single character or a word, corresponding to the smallest unit that can be inserted together with a reference to a preceding or succeeding W-Character.
- **v**: a boolean indicating whether the W-Character is visible. This is fundamental to the tombstone mechanism, as deleted elements remain part of the internal sequence while being hidden from the resulting document.
- id_{cp} : the identifier of the preceding W-Character.
- id_{cn} : the identifier of the succeeding W-Character.

Therefore, a W-Character can be represented as:

$$\langle id, \alpha, v, id_{cp}, id_{cn} \rangle \quad (1)$$

The use of vector clocks is fundamental for providing the **CvrdtDAG** history representation described in the following section 4.4.

A **WCharacter** with identifier **id** can be accepted by a **Site** under the following conditions:

- For an insertion, both id_{cp} and id_{cn} must already be present at the **Site**.
- For a deletion, the target **id** must already be present at the **Site**.

If a W-Character cannot be immediately accepted by the **Site**, the corresponding operation is stored in a pending-operation data structure. The pending operations are then checked whenever a new operation is received, allowing previously blocked operations to be applied once their required dependencies become available.

Another important limitation is the ever-growing number of tombstones resulting from deletion operations. This is a documented limitation of the original WOOT paper; the corresponding limitation and a possible solution are discussed in limit. item 13).

The core of the WOOT algorithm is the integration of W-Characters corresponding to insertion and deletion operations into a **Site**. This mechanism is fundamental to providing the **eventual consistency** guarantee of the data structure.

While deletion integration is straightforward (algorithm 2), as it only requires generating a tombstone by changing the visibility of the corresponding **WCharacter**, insertion integration is the central mechanism of WOOT (algorithm 1):

- Suppose an external W-Character c needs to be integrated into a **Site**. Its insertion operation guarantees that $ins(c_p \prec c \prec c_n)$, meaning that c must be placed between its previous and next W-Characters. However, other W-Characters may already have been inserted between c_p and c_n , for example as a result of concurrent operations. A deterministic procedure is therefore required to determine the position of c .
- First, the algorithm extracts from the sequence S the region relevant to the insertion of c , producing a subsequence S' bounded by its previous and next W-Characters. W-Characters outside this region do not affect the insertion position.
- If S' is empty, c can be directly inserted between c_p and c_n , since no other W-Characters are currently positioned between them.
- Otherwise, the algorithm identifies all direct overlapping competitors of c within S' , denoted as $d_0, \dots, d_m$. These are W-Characters whose previous and next references completely enclose those of c . The $\leq_S$ function is used to compare the positions of two W-Characters within the sequence.
- The sequence of competitors is then reconstructed as $L = [c_p, d_0, \dots, d_m, c_n]$. The algorithm iterates over this sequence until it identifies the appropriate position for c . A tie-breaking relation, $\leq_{id}$, is used to deterministically order concurrent competitors; in the original WOOT algorithm, this comparison was based on the site identifier and the operation's counter.
- Once the appropriate position has been identified, the integration procedure is recursively applied to the corresponding subregion, repeating the same process until c can be inserted into the sequence.

The rationale behind this procedure is to progressively narrow the interval in which c can be inserted. At each step, the algorithm uses the current pair of anchors to identify a smaller deterministic interval, repeating the process until only a single empty gap remains in which c can be placed unambiguously.

Algorithm 1 INTEGRATEINS(c, c_p, c_n)

Constraint: $S \in c_p, c_n$

```

1:  $S \leftarrow \text{sequence}$ 
2:  $S' \leftarrow \text{SUBSEQ}(S, c_p, c_n)$ 
3: if  $S' \neq \emptyset$  then
4:    $pos \leftarrow \text{POS}(S, c_n)$ 
5:    $\text{INSERT}(S, c, pos)$ 
6: else
7:    $L \leftarrow c_p d_0 d_1 \dots d_m c_n$ , where  $d_0, \dots, d_m$  are the  $W$ -characters in  $S'$ 
     such that  $C_P(d_i) \leq_S c_p$  and  $c_n \leq_S C_N(d_i)$ 
8:    $i \leftarrow 1$ 
9:   while  $i < |L| - 1$  and  $L[i] <_{id} c$  do
10:     $i \leftarrow i + 1$ 
11:  end while
12:   $\text{INTEGRATEINS}(c, L[i - 1], L[i])$ 
13: end if
```

Algorithm 2 INTEGRATEDEL(c)

Constraint: $S \in c$

```

1:  $c.v \leftarrow \text{False}$ 
```

The WOOT algorithm is implemented through the `ObservableState[CvRDTState, string]` and `CRDTOperation` abstractions, providing its functionality to the higher layers of the application. fig. 19 shows a simplified extract of the corresponding class UML diagram. For clarity, the UML block representing the `Site` has been omitted, as its implementation follows the algorithm described in the previous part of this section.

4.4 CvrdtDAG

The generic `History` implementation for a `Repository` provides two different ways of representing the operation history:

- An arbitrary linearization of the operations, sorted according to their `id-clock` pair.
- A DAG-based ASCII representation of the operations.

Under the hood, the `History` is implemented as `CvrdtDAGHistory`, which provides the following capabilities:

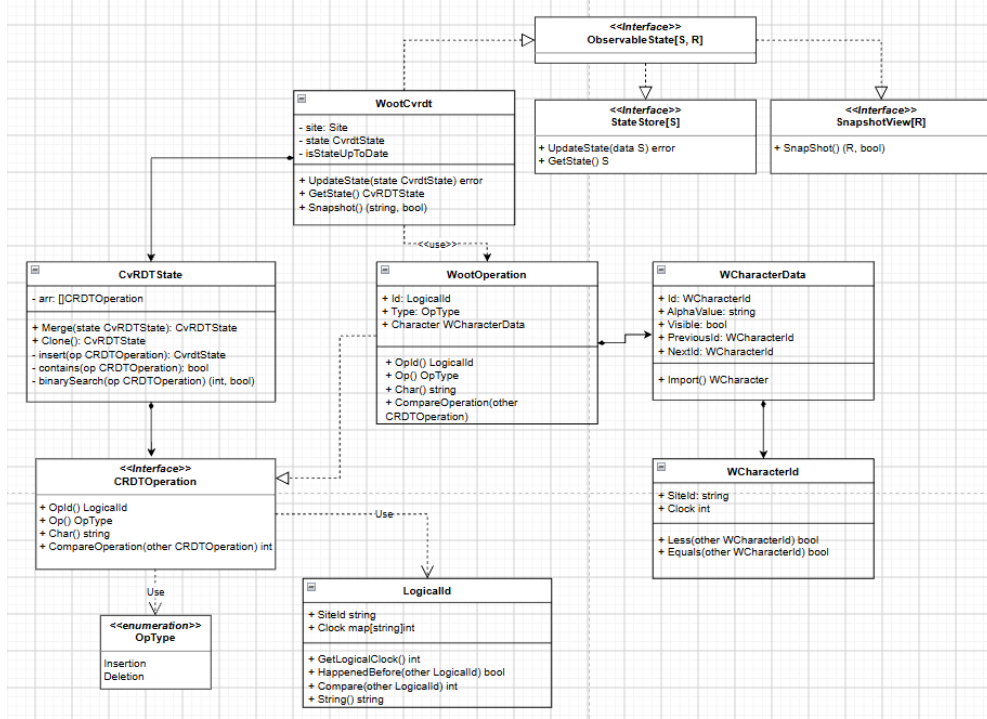


Figure 19: Integration of WOOT with design CRDT

- Each node in the DAG represents a **CRDToOperation**, without making assumptions about the underlying CRDT algorithm.
- The DAG is constructed from the happened-before relationship between operations. This is possible because every operation is associated with a vector clock.
- As previously discussed, each history representation is associated with a single file; a multi-file history representation is not provided.
- Each node can have multiple children, representing subsequent concurrent operations, and multiple parents, representing operations that depend on multiple concurrent branches. The resulting structure is a DAG because the happened-before relation is acyclic.

The **OperationDAG** is constructed from the transitive reduction of the partial order induced by the happened-before relation. This algorithm is not particularly efficient and may therefore become a limitation as the number of operations associated with a file increases. A possible improvement is discussed in limit. item 8.

The DAG is presented to the user as a topological, layered representation of the graph, where each layer n contains the operations whose dependencies belong to the preceding layers. Consequently, every operation, except those in the first layer, has at least one dependency in a preceding layer, while some operations may depend on multiple

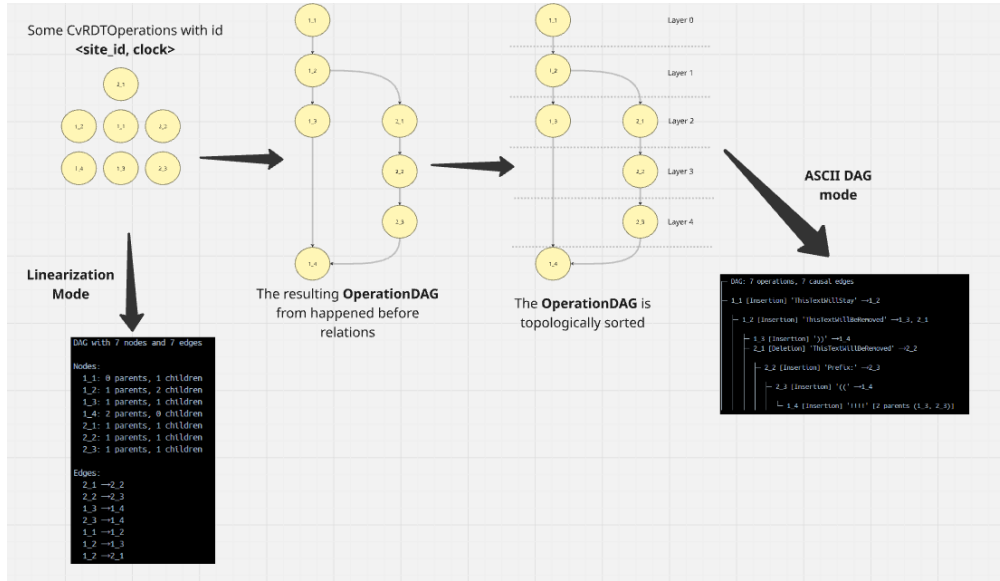


Figure 20: How history visualization works

operations from previous layers. The following fig. 20 illustrates the construction of a CvrDAG from a set of CRDTOperations and its corresponding visual representation.

5 Validation

As previously stated, the main goal of this application is to demonstrate the use of a DVCS in a decentralized setting through the adoption of CRDT-based states.

The validation process has two primary goals:

- verify that each component works correctly in isolation and that the individual components integrate correctly;
- validate the complete end-to-end system against the more demanding requirements of the application, particularly its non-functional requirements.

5.1 Automatic Testing

The first goal is addressed through unit tests. The second goal is more complex, as it requires the behavior of multiple distributed application instances to be evaluated as a whole. This is achieved by building end-to-end simulations in which multiple instances of the application interact as a distributed system.

These end-to-end tests can therefore be considered acceptance tests, as they validate the behavior of the complete system against its intended requirements.

5.1.1 Unit testing

These tests focus on different areas of the system. Some validate individual components, while others verify their integration with other components:

- **Peer tests:** they tests validate the low-level communication mechanisms between two nodes in the cluster (e.g., correct message reception and communication time-outs). Although they do not directly correspond to a specific requirement, they are fundamental for validating the requirements that depend on successful communication between cluster nodes (req. 1, req. 2, and req. 4).
- **Communication tests:** they validate the integration of all system components involved in higher-level communication, including low-level communication, session management, and remote peer management. Similarly to the **Peer tests**, but they provide further validation of the communication-related requirements.
- **Cvrdt tests:** these tests validate the CRDT module and the WOOT implementation, verifying both the correct merging of CRDT states across **Sites** and the correct creation of new CRDT operations. They are fundamental for validating req. 3 and provide the basis for validating the eventual consistency requirement (req. 1).
- **File tests:** these tests integrate the CRDT capabilities with the **File** module. They validate the correct behavior of CRDT-associated files and file-related operations, such as file context creation and file polling. They validate req. 5.
- **History tests:** they integrate the CRDT with **History** components. In particular, they verify the correct construction of the resulting **CvRDT DAG** from an underlying CRDT and whether the provided history representations (DAG and linearization) are correct. They validate req. 6.
- **Repository tests:** these tests integrate the **File** and **History** components to verify that they work correctly together as part of the **Repository**. They provide additional validation of the requirements covered by req. 5 and req. 6.

The unit tests can be run using the **make test** command. However, their intended use is to be executed automatically as part of the **convergent testing** process (section 5.2.1) when creating a new environment (section 6.1).

5.2 Acceptance test

While unit tests are essential for verifying individual components and their integration, they cannot fully demonstrate that the system behaves correctly as a whole. To address this, a simple and configurable end-to-end testing framework, called **convergent testing**, was provided.

A **convergent testing** automatically builds a simulated cluster and executes a configurable scenario against it. Its internal operation is described in greater detail in section 5.2.1.

Four tests were developed using this framework to assess requirements that are difficult to verify through unit testing, particularly non-functional requirements. These tests therefore serve as end-to-end acceptance tests for the system.

5.2.1 Convergent Testing

The unified framework implemented to support system-wide end-to-end testing is referred to as **Convergent Testing**.

A **Convergent test** is an end-to-end test that creates a cluster and simulates the execution of the software across multiple remote hosts.

The name derives from one of the main characteristics of these tests: at the end of each execution, the repository state of each peer is compared to verify that the CRDT-based DVCS reaches a common file state, even in a simulated distributed environment. This provides a means of validating the eventual consistency requirement of the system (req. 1).

Since the software is a proof of concept, this type of test also provides a convenient way to execute and validate the software:

- Executing a **Convergent test** configuration is the preferred way to run the software without relying on the cumbersome **Interactive mode** (section 6.2.1).
- Before building the simulated distributed scenario, all unit tests (section 5.1.1) can be automatically executed.
- Users can define their own configurations and run them as **Convergent tests**, allowing arbitrarily complex cluster scenarios to be simulated.

Despite this degree of flexibility, several tests were created as examples and to serve as software **Acceptance Tests**:

- **pairing_convergence test**: emulates a cluster with two hosts. One host joins the cluster first, while the second achieves convergence by receiving the CRDT state from the first.
- **complex_convergence test**: emulates a cluster with two hosts. Both hosts perform a larger number of operations, including insertions and deletions, and broadcast them to the other host at different times. Both hosts eventually converge to the same repository state.
- **peer_left_convergence test**: emulates a cluster with three hosts. The first peer creates an operation and broadcasts it to the other two peers before leaving the cluster. The remaining two peers then create a new operation. While the second and third peers achieve convergence with each other, the third peer does not converge with the other peers.
- **three_peer_convergence test**: emulates a cluster with three hosts. Each peer creates an operation and broadcasts it to the others at different times. All three peers eventually converge to the same repository state.

All of these tests validate important non-functional requirements in addition to the implicit req. 1:

- **Strong consistency for Cluster Membership** req. 2: this requirement is implicitly validated by all tests. Newly joining peers must successfully retrieve the consistent current cluster membership information from the **Cluster Server**. Failure to do so could also compromise the **eventual consistency** of the system.
- **Robustness to Communication Loss** req. 5: this requirement is exercised by several tests, especially the **three_peer_convergence test**. In these scenarios, multiple broadcast operations may be executed shortly before some peers effectively join the cluster, causing them to miss operation data. However, as long as the missing information is eventually broadcast again, the temporary loss of messages does not prevent the system from reaching convergence.
- **Fault Tolerance to Peer Failures** req. 3: this requirement is demonstrated by the **peer_left_convergence test**. A peer leaving the cluster during execution does not prevent the remaining peers from continuing to operate and eventually achieving convergence.

At this point, all non-functional requirements can be considered satisfied, with the exception of the portability requirement req. 4, which will be discussed in the next section.

In section 4.4, it was described how the operation history can be visualized either as a linearization, where operations are displayed sequentially according to simple tie-breaking rules, or as a DAG, which represents the classical history structure of a DVCS. In this implementation, the DAG is constructed according to the causal relationships between operations established by their vector clocks.

For each configuration, the **history** command (see section 7.1) can be used to dump a representation of the operation history, which should eventually converge across peers. Some of the resulting DAGs for the tests are shown in fig. 21:

Pairing convergence test : The DAG contains only one node, created by the first peer after joining the cluster (1.1 denotes the first operation created by the first peer to join the cluster), and no causal edges. The operation is subsequently pushed to the second peer after it joins the cluster.

Three peer convergence test : The DAG contains three nodes (operations) and two edges (causal dependencies). This means that each peer received the operation created by the previous peer before creating its own operation. All three peers perform an insertion operation, resulting in three sequentially ordered operations.

Peer left convergence test : The resulting DAG differs across the three peers. After creating the first operation (1.1) and pushing it to the other peers, the first peer leaves the cluster. The second peer then creates a new operation, which is subsequently pushed to the third peer. As a result, the three peers have different views of the operation history.

Complex convergence test : The DAG is more complex. As shown in the figure, after the second insertion performed by the first peer (1_2), both peers create independent operations before their histories are merged by a subsequent operation from the first peer. This does not cause a consistency problem, since the CRDT guarantees a deterministic representation of the resulting state regardless of the order in which concurrent operations are received.

6 Deployment

The installation of the software relies on the following dependencies:

- **GNU Make 4.4.1**: used to simplify and automate the Docker-based deployment workflow.
- **Docker 20.10.23**: the software is designed to be deployed using Docker. Docker Compose is not required. This choice was made to improve portability and satisfy the portability requirement (req. 4).

The fundamental unit of deployment is referred to as a **Peer Environment**. This term should not be confused with **CvrdtPeer**: a **Peer Environment** is a container hosting a complete application instance, which may contain both a **Cluster Client** and a **Cluster Server**.

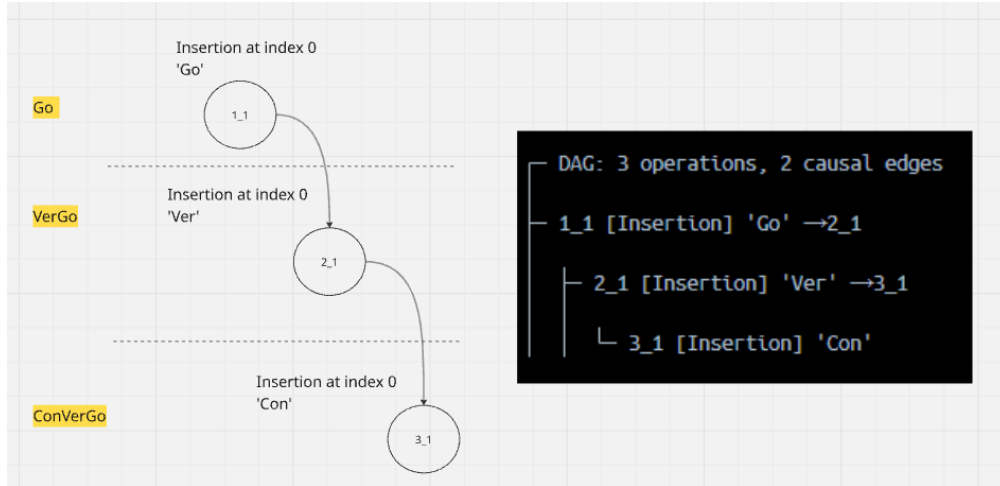
A **Peer Environment** consists of the following elements:

- A **lightweight Docker image**: the image provides all the capabilities required by the **Peer Environment** and is built automatically when needed. Its size is therefore kept as small as possible. Further information about its internal structure and build process is provided in section 6.1.
- A **hostname**: this provides a unique identifier for the container hosting the environment. The hostname is also used for domain name resolution, allowing the environment to communicate with other **Peer Environments**.
- A **Docker network reference**: this provides the network infrastructure required for communication between different **Peer Environments**.

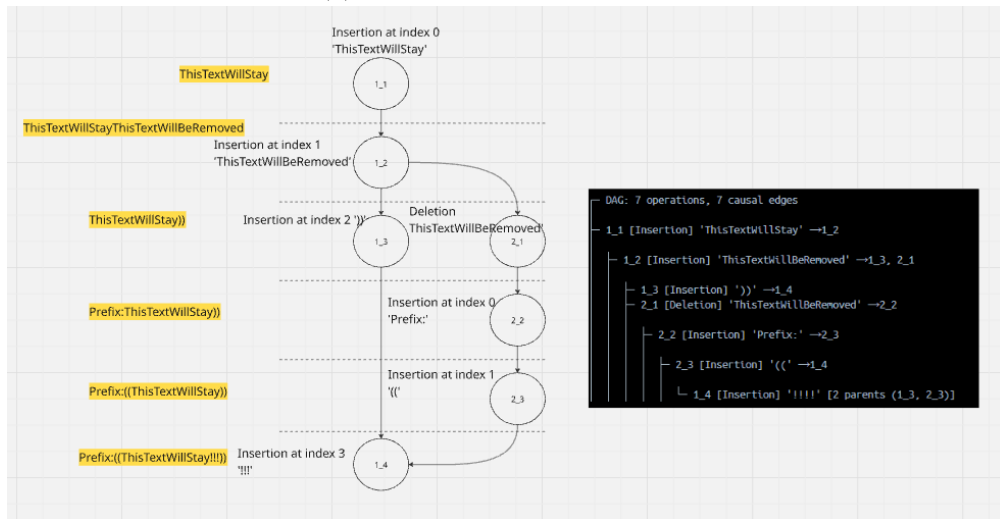
6.1 Docker environment

In the following subsection, the process used to provision a **Peer Environment** and its overall workflow are described. This information can also be derived from the project's **Dockerfile**.

The Docker workflow is implemented as a multi-stage build, which reduces the size of the final container image while keeping the different concerns of the build process separated. The resulting Docker build graph is illustrated in fig. 22. The stages are described as follows:



(a) three peer convergence DAG



(b) complex convergence DAG

Figure 21: Visual representation of convergent DAGs for tests

- **BUILD-STAGE:** this stage starts from an existing Docker image providing Go version 1.22.3 (`golang:1.22.3` on Docker Hub, an image based on Debian 12). It creates the project directory inside the container, downloads the Go dependencies, and compiles the project into a statically linked executable. Static linking avoids dependencies on external libraries during the subsequent stages and allows the executable to target a Linux environment.
- **TEST-STAGE:** this stage uses the same environment as **BUILD-STAGE** and executes all unit tests by invoking the appropriate Makefile command. This stage depends on the successful completion of the build stage.
- **BUILD-RELEASE-STAGE:** the image produced by the previous stages contains the complete build environment and its dependencies, which are unnecessary for running the application. Once the executable has been produced and the tests have passed, it is copied into a distroless container image, i.e., a lightweight image containing only the components required to execute the application. The resulting image can then be used to run the application.
- **DEV-RELEASE-STAGE:** this is an alternative stage in the build graph intended for development purposes. It is equivalent to **BUILD-RELEASE-STAGE**, but skips the testing stage, allowing the container image to be built more quickly during development.

6.2 Makefile Configuration

The Makefile is extensively used to simplify the interaction with the Docker configuration and provide better control over the deployment workflow.

The following are the most relevant Makefile commands:

- **make dock-build:** builds a Docker image named `$(IMAGE)` using the project's **Dockerfile**. As previously described, the resulting image is a lightweight distroless image. The unit tests are executed as part of this process. A development variant, **make dock-build-dev**, is also provided and skips the **TEST-STAGE** to allow faster builds during development.
- **make dock-net:** creates a virtual Docker network named `$(NETWORK)` (variable defined inside the Makefile), which is required to allow the different containers to communicate with each other.
- **make dock-clean:** removes the entire Docker infrastructure created by the project, including the Docker image, all **Peer Environment** containers (if in configuration mode), and the Docker network. In interactive mode it'll only clear the docker network and the image, the interactive container should be manually deleted

These three commands provide the fundamental operations required to set up and manage the deployment infrastructure.

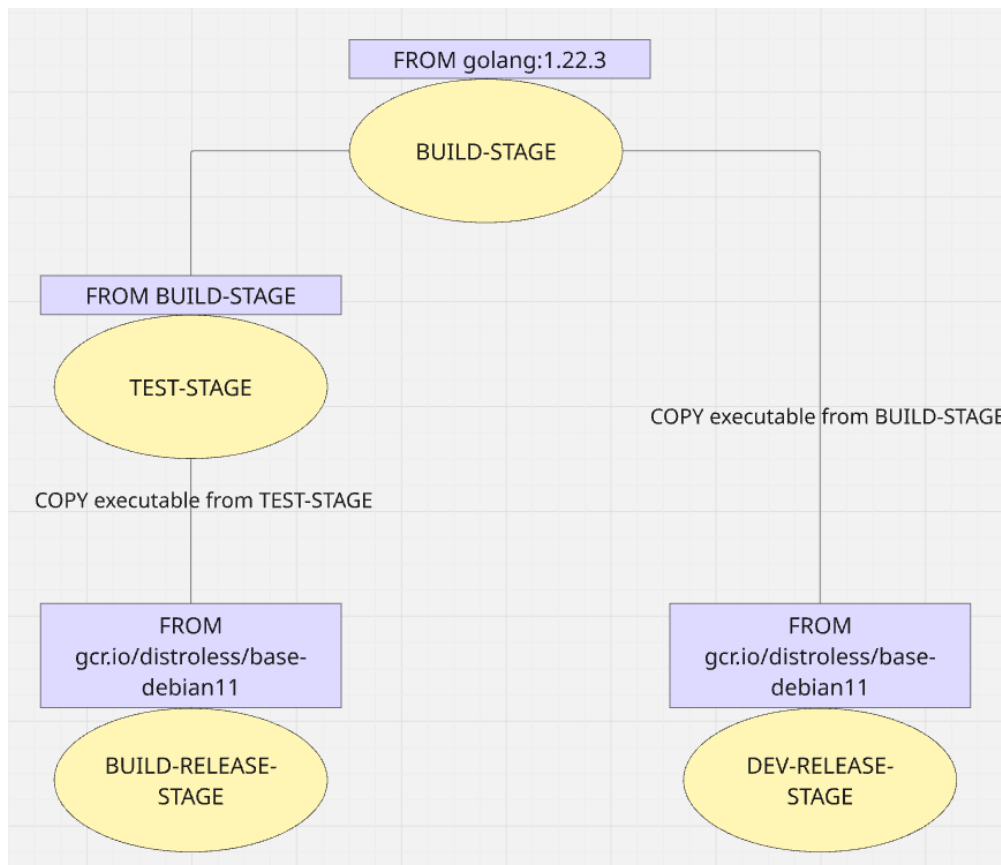


Figure 22: Dockerfile stage nodes deploy

At this point, it is necessary to distinguish between the two available execution modes, **Automated Configuration Mode** and **Interactive Mode**. The commands associated with each mode are described in the following subsections.

6.2.1 Interactive Mode

As previously introduced, **Interactive Mode** allows the user to interact directly with the console application and execute commands (see section 7.1 for an overview of all available commands).

Since the application is a proof of concept, this is not the preferred execution mode, as it requires several manual operations that make it cumbersome to use and test. In particular, the user must first create a **Cluster Server**, if one does not already exist, and then manually start the application multiple times to create multiple **Cluster Clients**. This is necessary when testing convergence or evaluating distributed scenarios involving multiple peers.

Even in this mode, the application is fully containerized. The **Peer Environment** has the following characteristics:

- The Docker image is run in an interactive container, with the container's **stdout** stream attached to the user's console.
- The user can specify the **Peer Environment** name through a command-line argument. This name is subsequently used to identify the environment and communicate with it from other containers.

The following commands are provided specifically for the interactive mode:

- **dock-test-int NAME_INT=<envname>**: creates a **Peer Environment** after executing the unit tests. The user must provide **<envname>**, which is used as the environment's hostname and identifier for communication with other **Peer Environments**. As with the other execution modes, the environment is connected to the Docker network specified by the Makefile's **\$(NETWORK)** constant.
- **dock-test-dev-int NAME_INT=<envname>**: equivalent to the previous command, but skips the testing stage, allowing the environment to be created more quickly during development.
- **dock-dump PEER_NAME=<envname>**: outputs all of the container logs till now to **stdout**

6.2.2 Configuration mode

This deployment mode constitutes the core of the **Convergence Testing** workflow described in section 5.2.1.

In this mode, a list of **Peer Environment** is created automatically. Each for them reads from a dedicated command file containing a list of newline-separated commands.

This allows multiple environments to be created and configured without requiring manual interaction with each container.

The main characteristics of this mode are:

- Each container is deployed in detached mode, allowing the configuration process to continue without requiring an interactive console for each environment.
- Each container is removed when a new configuration is started, but not when the current execution terminates. This allows the user to inspect the containers and their state after the execution has completed.
- The mode is primarily intended to test convergence between multiple **Peer Environments** and is therefore closely integrated with the **Convergence Testing** workflow. However, it can also be used to automate the execution of specific cluster simulations without requiring manual interaction in production scenario.
- The system supports up to 10 **Peer Environments** within a single simulation by default. This limit can be increased by modifying the `$(MAX_PEERS_NUM)` constant in the Makefile.

Unlike the previous mode, this is the preferred approach for testing the software. It allows users to easily define their own cluster configurations and evaluate the system under increasingly complex scenarios, while verifying that the resulting state of the peers converges correctly.

A new configuration can be created by adding a directory named after the configuration inside the `./test_convergent/` directory. The internal structure of this directory should follow the structure shown in fig. 23. More specifically:

- The configuration must contain one command file for each **Peer Environment** that will be created. Each file contains one command per line (see section 7.1 for the list of available commands).
- Each command file must follow the naming convention `peerN.txt`, where N identifies the peer and corresponds to its position within the testing configuration.
- The user does not need to explicitly create a dump directory. This directory is created automatically and is used to store the complete container logs for each **Peer Environment** after each execution. A timestamp is included in the directory name to facilitate the identification and analysis of individual test executions.
- The `.env` directory allows the user to define environment variables. The `peerN.env` files contain variables specific to the corresponding **Peer Environment**, while `global.env` contains variables shared by all environments.

Environment variables can be directly interpolated within command files using the `${...}` syntax. This mechanism simplifies the definition of complex commands by allowing values to be specified once and reused across multiple commands. In addition to the variables defined by the user, some environment variables are automatically injected into each **Peer Environment**:

- `$DOMAIN`: defines the domain name used to identify the **Cluster Server** within the simulation. Its value can be configured in the Makefile through the `$(DOMAIN_SERVER_PATH)` constant.
- `${TARGET_FILE}`: identifies the file that is modified during the simulation and used at the end of the execution to verify convergence. Its value can also be configured through the Makefile.
- `${COMMAND_FILE}`: contains the path of the command file from which the commands are read. This variable is injected programmatically into each **Peer Environment**.
- Additional environment variables can be defined by the user through the local `peerN.env` files or the global `global.env` file.

Once the configuration has been defined, it can be executed using the following command, specifying both the maximum number of peers and the name of the test configuration (see the user guide in section 7):

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

An important aspect of this command is that `N` represents the maximum number of peers to be used, rather than the exact number of peers defined in the configuration. This provides additional flexibility when defining test configurations, as the same configuration can be executed with fewer peers than originally specified.

For example, given a configuration containing `peer1.txt`, `peer2.txt`, and `peer3.txt`, executing the test with `N=2` creates only the first two **Peer Environments**, while `peer3.txt` is ignored.

The final convergence of the cluster can then be evaluated. To perform this evaluation correctly, two conditions must be satisfied when defining the commands contained in each configuration file:

- The peers must modify the `$TARGET_FILE` file. This file represents the repository file used to verify convergence.
- A **Cluster Server** must be created using the same domain name specified by the `${DOMAIN}` variable. The corresponding server path is defined by the `$(DOMAIN_SERVER_PATH)` constant in the Makefile, which identifies the directory where `${TARGET_FILE}` is stored inside each container.

Once the execution is completed, the files located at `peerN:$(DOMAIN_SERVER_PATH)/$TARGET_FILE` are compared across all participating containers. If all files contain the same content, convergence is considered successful, indicating that all cluster members have eventually reached the same CRDT state.

7 User Guide

The way the software can be executed and the list of available cluster simulation are available in the README.md.

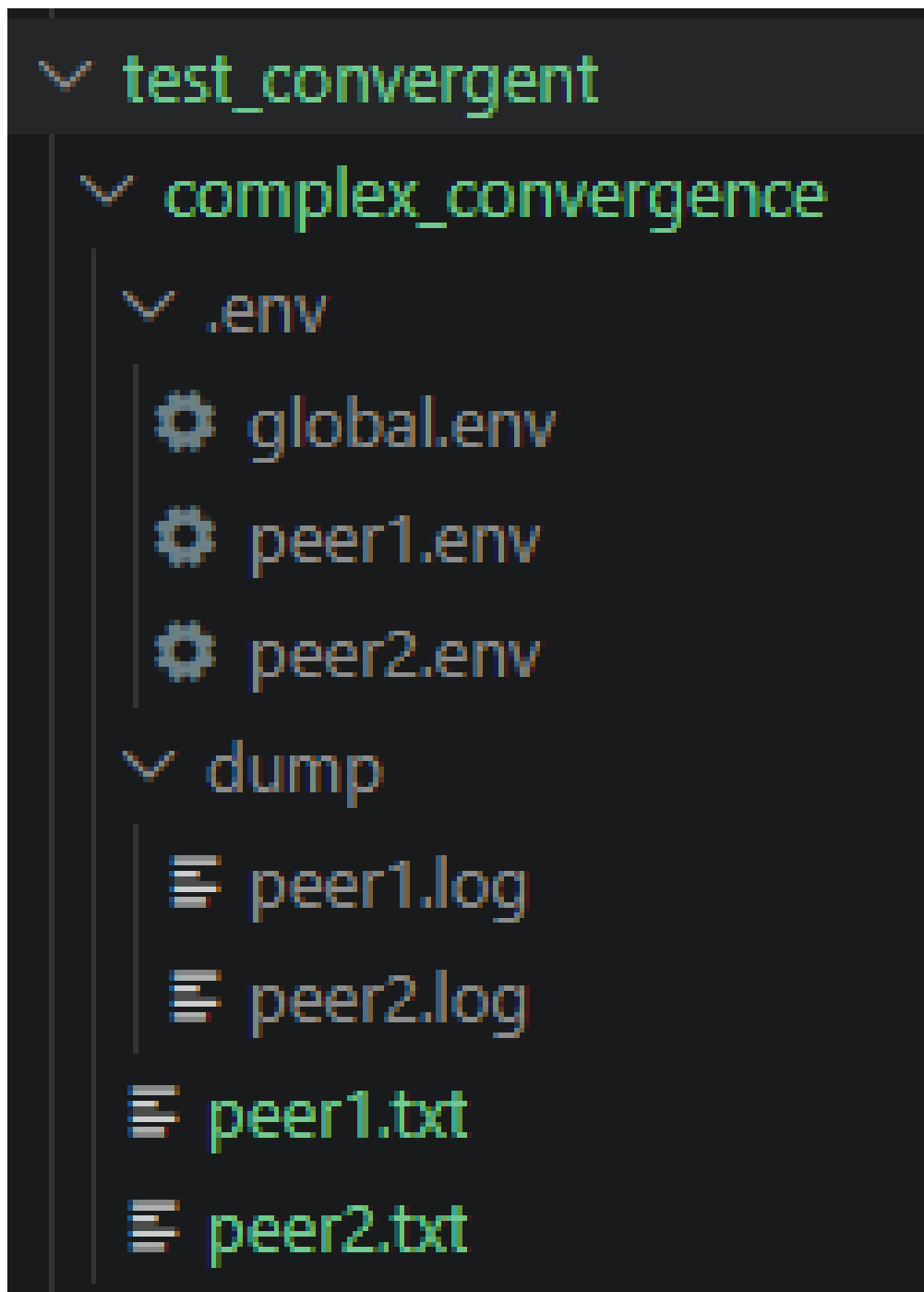


Figure 23: Configuration example structure for a custom convergent testing

The user is encouraged to first execute the simulations already provided in configuration mode, as described in the README.md. After that, if necessary, they can experiment with interactive mode or create custom configurations.

Run in configuration mode (unit tests will also be executed)

```
make dock-test N=<num_env> TEST_DIR=<test_name>
```

Run in configuration mode (unit tests will not be executed)

```
make dock-test-dev N=<num_env> TEST_DIR=<test_name>
```

Run a single node environment in interactive mode (unit tests will also be executed)

```
make dock-test-int NAME_INT=<envname>
```

Run a single node environment in interactive mode (unit tests will not be executed)

```
make dock-test-dev-int NAME_INT=<envname>
```

7.1 List of commands

edit insert <filename> <pos> <text> : Inserts **text** into **filename** in the repository. If **pos** is greater than the maximum position, it is rounded to the highest valid position. If successful, a new operation is created in the repository.

edit delete <filename> <pos> : Deletes the character at position <pos> in **filename** in the repository. If **pos** is greater than the maximum position, it is rounded to the highest valid position. If successful, a new operation is created in the repository.

push all : Broadcasts all currently owned operations in the repository to the other peers in the cluster.

read <filepath> : Prints the text contained in **filename**.

history <filepath> : Print a linearization history of all the **filename** operations.

history <filepath> <maxdepth> : Print a DAG history of all the **filename** operations (the **maxdepth** parameters provide an upper-bound to the displayed DAG topological layers)

create_cluster_client <clientInfo> <domainName_address> : Creates a new **Cluster Client** by providing a **clientInfo** and binding it to a **Cluster Server** domain name and address.

`create_cluster_server <domainName_address>` : Creates a new Cluster Server by providing its domain name and address.

`connect_to_cluster <peeraddress>` : Initializes a connection between the Cluster Client and the Cluster Server. It also provides the address that will be used by the CvrdtPeer.

`shutdown <nodeType>` : accept as argument "client" or "server", shutdown completely the current default node (client or server) and disconnect from others

`shutdown_app` : the whole application is shutted down

`wait <millis>` : Delays execution for the specified number of `milliseconds`. This command is useful only in configuration mode to emulate pauses.

8 Self-evaluation

I must admit to have mixed feelings about my work. Although I would not necessarily consider this a negative aspect, I believe that I significantly scaled back some of the ideas I had originally intended to implement, ultimately limiting the scope and potential of the final result, while still remaining within the proposed scope and requirements.

Another challenge was the fact that I had to interrupt development for several months during the project. This resulted in the software evolving considerably differently from the initial plan, mainly because I was unable to dedicate a sufficient amount of time to its development in a continuous manner. Nevertheless, the final contribution substantially respected the initially planned 90 hours of work.

My original objective was to explore and experiment with the integration of collaborative CRDT technologies into a DVCS. I have always found these systems particularly interesting, while also recognizing that integrating them correctly is far from trivial, as demonstrated by the numerous limitations that I ultimately had to identify in my work. From a learning perspective, however, I believe that the development process allowed me to explore these technologies in considerable depth and to confront several non-trivial aspects of distributed systems.

In retrospect, I should have allocated more of the available time to further distributed-systems aspects that could have provided greater value to the project, such as stronger delivery semantics, smarter transport management, improved deployment mechanisms, or even a truly decentralized consensus-based system. Instead, I did not always make the most effective use of the available 90 hours, at times focusing on more peripheral aspects of the software that ultimately had a relatively limited impact on the final result.

Looking back, I do not think I would reproduce every aspect of what I actually developed if I were to undertake the project again. Nevertheless, I believe that I was able to respect what I described in the proposal, and, most importantly, remain within the minimum scope I had originally defined for the project.

9 Future works

The developed system is far from being production-ready. For this reason, the following subsection discusses the main limitations of the current implementation. These limitations range from relatively minor areas for improvement to significant design trade-offs and potential new features that could be introduced in future versions.

10 Limitations

1. **CvRDT over CmRDT:** CRDTs can be implemented as CvRDTs, which exchange replica states, or CmRDTs, which exchange operations. CvRDTs simplify synchronization but may require transmitting a growing amounts of state. CmRDTs reduce communication overhead by propagating only operations, but require additional mechanisms for message arrival assurance. Thus, CvRDTs favor simplicity, while CmRDTs favor communication efficiency. Despite Woot being a CmRDT, it was wrapped into a CvRDT, where the CRDT state consists of all the internally stored operations. Implementing the CRDT directly as a CmRDT would provide more efficient communication, but would exceed the project's 90-hour time constraint. A slightly simpler alternative would be to adopt a hybrid approach, such as using a Delta-State CRDT to avoid transmitting the entire state for every synchronization.
2. **At-most-once over exactly-once delivery semantics:** Adopting a weaker delivery semantic is a direct consequence of using a CvRDT. Since synchronization is based on exchanging states rather than individual operations, losing a state message does not compromise eventual consistency: a subsequent synchronization can deliver a more recent state that incorporates the missing updates. Therefore, there is no strict need for retransmission or exactly-once delivery semantics, allowing the system to rely on at-most-once delivery.
3. **No Docker Compose deployment:** Using a more structured and powerful tool for multi-container orchestration, such as `Docker Compose`, instead of directly managing containers with `Docker`, could have provided a simpler and more convenient approach for orchestrating `Convergent Testing` section 5.2.1. It could also have reduced the complexity of the user-written `Makefile`.
4. **Cluster Server as a single point of centralization:** This is the most important limitation of the system, as it was explicitly identified in the original project proposal. A failure or shutdown of the `Cluster Server` would prevent not only new clients from joining the cluster, but also the periodic synchronization with the server that currently takes place before each data broadcast.

A simple way to mitigate the impact of a server failure on peers that are already part of the cluster, although not on clients attempting to join, would be to make the `Cluster Server` proactively notify peers whenever another peer disconnects

from the cluster. This improvement would require only a small amount of additional work and would also make the system more agile while reducing the load on the server, since peers would no longer need to contact the server before every broadcast.

Achieving a fully decentralized system would require considerably more work. In particular, a cluster consensus mechanism would be needed to elect a new server when the current one leaves or fails. Alternatively, to reduce the load on a single node, multiple nodes could be elected to handle requests. However, this approach would introduce additional consistency challenges, as maintaining strong consistency across multiple request-handling nodes would be more difficult in the presence of network partitions, in accordance with the CAP theorem.

5. **Linear state growth in terms of operations:** Broadcasting the `Repository` content as a `CvRDT` also means that the amount of data transmitted grows over time as the number of operations increases. Since the system does not implement an explicit delivery-tracking mechanism, it cannot determine which operations are already known by a peer. Consequently, the entire state must be broadcast at each synchronization. This limitation has no straightforward solution without changing the underlying design, as it directly results from the combination of using a `CvRDT` and adopting an `at-most-once` delivery semantic.
6. **Usage of vector clocks and garbage collection:** In the original Woot implementation, each CRDT operation is identified by a pair $\langle siteId, logicalClock \rangle$. While the paper does not strictly require a logical clock as the identifier (a counter was enough), a logical clock is more than sufficient to uniquely identify operations within a site.

In the provided implementation, however, the Lamport clock is not sufficient to satisfy the causal history requirement, as it cannot represent a partial order or distinguish concurrent events from sequential ones (i.e. some concurrent event could be classified as sequential wrongly). Therefore, vector clocks were adopted to explicitly represent causal relationships between operations.

This choice introduces a scalability trade-off: the size of the vector clock associated with each operation grows with the number of replicas in the cluster. Moreover, in the current implementation, the vector clock does not only grow with the number of currently active replicas, but also retains entries for replicas that previously joined and subsequently left the cluster, as no vector clock garbage-collection mechanism is implemented.

A distributed mechanism for safely removing obsolete vector-clock entries could be implemented, but this was considered outside the scope of the project. The additional metadata overhead was therefore accepted as a trade-off to satisfy the causal history requirement within the project's constraints.

7. **Ping Server Problem** The need to continuously ping the server before each peer broadcast is an important feature for preventing cluster partitions. However, this

approach places excessive load on the **Cluster Server**, with the strain increasing as more peers need to share their repository state.

An alternative approach could be to implement a lightweight heartbeat mechanism that requests information from the server at a configurable periodicity. This would also allow the user to control the maximum window of possible inconsistency in the system. The heartbeat mechanism could further be implemented using a lightweight UDP transport protocol to reduce the load on the server. This would require providing a **Transport** implementation based on UDP.

8. **DAG building inefficiency** The DAG construction algorithm iterates over all operations defined for a file. For each pair of operations, the happened-before relation is evaluated to determine whether an edge should be added. When a happened-before relationship is found, an additional iteration is performed to determine whether an intermediate operation exists that makes the relationship transitive. Consequently, the worst-case time complexity is $O(n^3)$, where n is the number of operations, while the auxiliary space complexity is $O(1)$.

The algorithm can be optimized to achieve $O(n^2)$ time complexity by avoiding the explicit search for an intermediate operation for every happened-before pair and instead caching the required relationship information for reuse. This optimization introduces an additional space requirement, potentially increasing the auxiliary space complexity to $O(n^2)$ in the worst case.

While the current implementation is sufficient for datasets containing approximately $n = 100$ operations, the optimized approach may be preferable when dealing with substantially larger numbers of operations.

An even more effective optimization would be to avoid reconstructing the entire DAG whenever it is requested. Instead, newly created operations could be incrementally added to the existing DAG, updating only the affected nodes and edges. This approach would avoid repeatedly processing the complete set of operations and would provide better scalability as the number of operations grows.

9. No editor support for insertion and deletion

As the application is implemented as a simple CLI, it does not provide integration with a text editor or any graphical user interface. This can make it cumbersome for users to identify the exact position at which text should be inserted or deleted, particularly when working with larger files.

Although providing editor integration is outside the scope of this project, it would represent a useful enhancement for improving the overall usability of the application.

10. No Navigable History

The **Repository** module does not provide a mechanism for navigating the history of a file, such as changing the current head to a specific operation ID and reconstructing the corresponding file content at that point in the history.

This feature could have been implemented by making relatively minor modifications to the WOOT implementation. For example, a temporary site could be created to apply the inverse of selected operations without modifying the original repository state. This could enable alternative forms of history navigation, such as moving the head to a specific operation or reconstructing the state corresponding to a particular set of concurrent operations at a given point in the history.

11. No Pull Approach

From a distributed-systems perspective, the current implementation adopts a fully push-based data dissemination model. Every peer that produces a data flow (i.e., broadcasts its repository operations) is also responsible for driving that flow, while receiving replicas simply consume the incoming data.

This approach generally provides lower propagation latency, since communication can start as soon as a new operation needs to be disseminated. However, it presents some scalability limitations compared with a pull-based approach. In particular, with push-based dissemination, every replica is actively sent each operation and must process the corresponding message, even when the update may not be immediately required. A pull-based approach would instead allow peers to request updates only when needed, potentially reducing unnecessary communication and processing.

A pull-based synchronization mechanism could have been implemented as an alternative. The system already employs a similar approach when a client requests cluster membership information from the server. Moreover, supporting both push and pull mechanisms would have made the system more closely resemble traditional DVCSs, where synchronization can involve both pushing local changes and pulling remote changes.

12. **No form of authentication:** No real form of authentication is provided by the system: the client can declare a username, assign an address, and be assigned an ID for its created `CvrdtPeer`, but this data can be easily forged and no password-based authentication is performed. Moreover, there is no way to log out from the cluster and re-enter it using previously created credentials. This feature could be implemented by providing the `Cluster Server` with a password store (e.g. hashed passwords) and validating incoming credentials along with the user data.
13. **No garbage collection for WOOT 'tombstones':** In the original WOOT implementation, deletion operations work by marking elements as invisible so that they are not displayed in the sequence. This mechanism is called **tombstone marking**, meaning that even a small text sequence can eventually contain a large number of tombstones with no practical value. To garbage collect a tombstone, it would be necessary to ensure that every current peer in the cluster has already received it. At this point, the decentralized cluster could collectively agree to remove the corresponding operation. Implementing this approach is not a simple matter, and it was also explicitly identified as a limitation in the original paper.

14. **No loosely coupled MOM:** Instead of using a `Session` layer over the `Transport`, a way to reduce the load on individual clients could be to introduce a completely separate node acting as a MOM. This would also make it considerably simpler to define stronger delivery semantics without overcomplicating the code, especially when using existing MOM frameworks such as `RabbitMQ`, `Kafka`, or `Redis` for a highly scalable system.
15. **Multiple clients/servers on the same instance** The `ClusterNodeStore` is the main component of the system that can potentially host multiple clients and servers on the same machine. In the current implementation, however, only one client and one server per node can be hosted. While hosting multiple `Cluster Servers` and `Cluster Clients` on the same machine would be possible as a feature, it was not implemented for simplicity.
16. **More space-efficient serialization** While de/encoding messages using json is simpler, binary serialization formats such as `GRPC` with `Protocol Buffers` can provide better space efficiency than `JSON`. They avoid textual representations and encode values using compact binary representations, reducing message size and transmission overhead while generally providing faster serialization and deserialization.
17. **Operations over commits** The software provides no notion of commits: the idea was to focus on a smaller-granularity entity, namely individual operations. Visualizing the history DAG in terms of operations rather than blocks of them (commits) also makes the history more comprehensible. Moreover, this approach better unifies the DVCS notion with the collaborative approach, where individual operations play a central role.
18. **Transport Bytestream Size Limitation**

To support variable-size messages, such as broadcasts of the repository state, a size header encoded in big-endian format is prepended to each serialized message before transmission. To prevent the bytestream from growing arbitrarily, a maximum message size of 4MB is enforced.

Once this limit is exceeded, the system can no longer properly broadcast the growing repository state. This does not cause a fault or fatal error in the cluster; rather, messages exceeding the limit simply cannot be transmitted.
19. **File History over Repository History**

The history is implemented at the level of individual files rather than the repository. This choice was mainly made to simplify the system and make its behavior easier to validate and more idiomatic considering the adopted CRDT algorithm.

Since WOOT is a collaborative editing CRDT designed around a single sequence (i.e., the content of a file), supporting repository-level history would require some modifications to the current implementation. However, this could be achieved

without fundamentally changing the system, for example by associating operations with files and maintaining a higher-level history across the repository.

20. **Theoretical UUID collision:** UUIDs are used as global identifiers to distinguish message sessions over a specific topic. Although the probability of generating the same UUID for two different sessions is extremely small, it is not zero. In UUID version 4, 6 of the 128 bits are reserved for the version and variant fields, leaving 122 bits available for random generation. Therefore, the total number of possible UUIDs is:

$$2^{122} \approx 5.3169 \times 10^{36} \quad (2)$$

On the other hand, the local ID approach eliminates the need for session ID synchronization, reducing both transmission overhead and system complexity.

References

- [1] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [2] Paul J. Leach, Rich Salz, and Michael H. Mealling. A Universally Unique Identifier (UUID) URN Namespace. RFC 4122, July 2005.
- [3] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work (CSCW '06)*, pages 259–268. ACM, 2006.
- [4] Hector Sanjuan, Samuli Poyhtari, Pedro Teixeira, and Ioannis Psaras. Merkle-crdts: Merkle-dags meet crdts. *CoRR*, abs/2004.00107, 2020.