Distributed Multi-Warehouse Inventory Manager

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

Enrico Cornacchia
enrico.cornacchia@studio.unibo.it

September 10, 2026

This report presents the design, implementation, and validation of a distributed, multi-warehouse inventory management system. Built to operate across a simulated Wide Area Network, the system uses a masterless, peer-to-peer architecture that prioritizes Availability and Partition Tolerance over strict consistency. The core objective is to ensure that local warehouse operators can continuously read and modify stock levels without blocking, even during severe network outages. The backend nodes are implemented in Go and communicate asynchronously via gRPC, utilizing embedded SQLite databases for serverless, lightweight data persistence. In order to guarantee eventual consistency and prevent data loss during split-brain scenarios, the system employs vector clocks for causality tracking and an additive state reconciliation protocol. The infrastructure is containerized using Docker and orchestrated via a local Kubernetes cluster (Minikube). Furthermore, Project Calico is integrated as the Container Network Interface (CNI) to enforce network policies, allowing for rigorous testing of simulated network partitions. The final system successfully demonstrates fault tolerance, automatic state recovery, and mathematical conflict resolution without relying on a central authoritative database.

Disclaimer

During the preparation of this report, the author used LLMs to refine syntax and correct linguistic errors. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of the final report.

1 Concept

This project mainly consists of a decentralized backend web service composed of independent peer nodes, accompanied by a Command-Line Interface (CLI) client.

- **Intended Users**: The main targets of this application are the warehouse operators and inventory managers who require to track stock levels across a multi-location retail business.
- **Use Case**: Throughout the operational day, workers use the system to record stock deductions (like local sales) and additions (incoming shipments). Since the system prioritizes availability, users can always read and write to their local warehouse node, even if the connection to the rest of the enterprise network drops.
- **Interactions**: The users interact through a local CLI, submitting inventory changes via text commands. While user inputs are manual, background interactions between the nodes happen continuously via gRPC network calls to synchronize the global inventory state and resolve conflicts.
- **Roles**: The system features two main components: the CLI client and the peer node. The client acts as a simple interface for the user, while multiple peer nodes interact with each other as equals to maintain and merge the distributed ledger.
- **Data**: The system persistently maintains data related to the inventory, specifically item IDs and stock quantities, along with vector clock states required to accurately resolve concurrent updates. This data is permanently stored on the local disk of each node using an embedded SQLite database.
- **Geographical Distribution**: The system is designed to operate across a WAN, simulating a company with multiple physical warehouses located in different cities. Because internet outages can occur over large distances, the system is explicitly built to tolerate network partitions, eventually synchronizing states once the connection is reestablished.

2 Requirements Elicitation and Analysis

This section defines the functional and non-functional requirements of the system, as well as the implementation constraints for it.

Functional Requirements

1. **Local Stock Modification**: The system must allow a user to submit a quantity change for a specific item via the CLI. **Acceptance criterion:** The user inputs an item ID and a delta value, and the system immediately reflects the new total on the local node.

2. **Local Inventory Read**: The system must allow a user to query the current stock level of a specific item or the entire catalog. **Acceptance criterion:** The user executes a read command in the CLI, and the system outputs the current stock quantities stored in the local ledger.

3. **Peer Synchronization**: The system must automatically synchronize its local ledger with other available nodes in the network without manual user intervention. **Acceptance criterion:** When two nodes are connected to the network, an update on Node A is eventually propagated to Node B, and both nodes reflect the same global inventory state.

4. **System Audit**: The system must provide a mechanism to view the internal metadata of a node, specifically its vector clock history, for debugging and auditing purposes. **Acceptance criterion:** Using the `-get` (single item) or `-all` (full ledger) CLI flags prints both the inventory quantities and the raw vector clock state of the local node to the terminal.

Non-Functional Requirements

1. **High Availability (AP part of CAP)**: The system must prioritize availability over strict consistency. A node must accept local read and write operations even if it is completely disconnected from the rest of the network. **Acceptance criterion:** During a simulated network partition, CLI commands to modify and read stock on an isolated node succeed without timing out or throwing network errors.

2. **Eventual Consistency and Conflict Resolution**: The system must correctly reconcile concurrent updates once a network partition heals, ensuring no data is lost. **Acceptance criterion:** If Node A and Node B concurrently deduct 5 units from an initial stock of 100, the final synchronized state across both nodes must be 90, correctly reflecting both deductions.

3. **Usability**: The application should provide a straightforward and intuitive command-line interface for the end user, abstracting away the complexities of the underlying distributed network. **Acceptance criterion:** A user with basic terminal knowledge can execute inventory commands without needing to understand gRPC or peer-to-peer networking.

4. **Performance and Throughput Trade-offs**: Local operations do not require network consensus and should be highly responsive to the human operator. **Acceptance criterion:** Under normal human-operator load, local CLI reads and writes return in under 100 milliseconds. However, because local thread safety

strictly serializes operations to protect the file system, the system accepts a capped throughput in exchange for absolute data integrity.

5. **Thread Safety and Local Data Integrity**: Because the gRPC server and background gossip daemon operate concurrently, the local node must prevent read-modify-write race conditions. **Acceptance criterion**: Concurrent CLI requests and incoming network syncs are safely serialized using application-level locks and strict database connection limits, ensuring no local data corruption occurs even under simultaneous load.

Implementation Requirements

1. **Architecture**: The system adopts a decentralized, leaderless peer-to-peer architecture for the warehouse nodes, accompanied by a local client-server model for interactions between the CLI and its host node. This ensures fault tolerance and eliminates single points of failure. **Acceptance criterion:** The system operates entirely without a central orchestrator or master node.

2. **Programming Language**: The backend nodes and CLI must be developed using Go, as it provides excellent native support for gRPC and powerful concurrency primitives (goroutines/channels), which perfectly match the system's need to handle concurrent peer-to-peer network calls. **Acceptance criterion:** The codebase compiles natively using the standard Go toolchain.

3. **Persistent Storage**: The local ledger for each node must be stored using an embedded SQLite database. Since the architecture removes the central database to eliminate single points of failure, each node requires lightweight, serverless persistence that can easily survive container restarts. **Acceptance criterion:** A node can crash, restart, and perfectly recover its inventory state and vector clock history from a local `.db` file.

4. **Containerized Orchestration**: The deployment must be orchestrated using Docker and Kubernetes (via Minikube). This way, it is possible to realize the distributed orchestration and simulate a WAN environment locally. **Acceptance criterion:** The system can be entirely spun up using `kubectl apply` commands on a local Minikube cluster.

2.1 Relevant Distributed System Features

Follows a list of the relevant distributed systems features related to this project:

- **Fault Tolerance and Dependability**: This is the primary driver of the architecture. Uninterrupted service is strictly required at the local warehouse level. The system handles component failures by isolating them; if one node crashes, it does not impact the availability of others.

- **Resource Sharing and Coordination**: The global inventory is a shared resource. Because there is no central lock manager, coordination is achieved mathematically using Vector Clocks to track causality and resolve concurrent modifications safely.
- **Performance and Concurrency**: Background synchronization relies on parallel network requests to gossip state without blocking the user's CLI interactions. Locally, strict concurrency controls are implemented to safely process simultaneous requests from the CLI and incoming peer syncs without corrupting the local SQLite ledger.
- **Transparency**: The system implements location and replication transparency from the user's perspective. The warehouse operator simply interacts with their local CLI, completely unaware of the background gRPC synchronization or the physical locations of other nodes.
- **Scalability**: The peer-to-peer nature allows new warehouse nodes to join the cluster. The decentralized design prevents a single database from becoming a bottleneck as the enterprise grows.
- **Economy and Costs**: By leveraging a decentralized architecture with SQLite, the system avoids the financial overhead of provisioning and maintaining a highly available central database cluster.
- **Security and Trust**: The current prototype omits authentication and authorization mechanisms. Because the application strictly processes internal stock quantities rather than sensitive personal data, strict access control is not a critical requirement for this phase of development.

3 Design

This chapter explains the strategies used to meet the requirements identified in the analysis.

3.1 Architecture

The system adopts a masterless, decentralized Peer-to-Peer architecture. Traditional centralized databases create single points of failure and latency bottlenecks over a Wide Area Network. By ensuring every warehouse node operates as an equal peer with its own local ledger, the system explicitly prioritizes Availability and Partition Tolerance. This guarantees that local read and write operations never block, even during severe network outages.

3.2 Infrastructure

The topology consists of two primary components: a lightweight CLI client (acting as the user interface) and the Warehouse Peer Nodes (acting as both the application server and the data persistence layer). There are no central load balancers, message brokers, or external databases.

In production, these nodes would be geographically distributed across distinct physical warehouses. For this prototype, they are distributed as isolated pods within a local Kubernetes cluster. Components locate one another using Kubernetes' internal DNS and Service routing. The CLI client connects to its local host node via localhost port forwarding, while the peer nodes discover one another via static Kubernetes service names, removing the need for a complex external service registry.

3.3 Modelling

The core domain entities are **Inventory Items** (identified by a string ID), **Stock Quantities** (integers), and **Vector Clocks** (a map of node IDs to logical timestamps).

These domain entities map directly to the Warehouse Peer Nodes. The global state is fundamentally partitioned and replicated; each node holds a complete, eventual copy of the inventory data mapped directly to its local disk via an embedded SQLite database.

The system exchanges two types of messages over gRPC: **Commands** (sent from the CLI to the local node to mutate state or query data) and **Events** (gossip payloads exchanged between peer nodes containing their updated vector clocks and inventory ledgers).

3.4 Interaction

The system relies on asynchronous, bidirectional communication to maintain both local availability and global consistency. The interactions are divided into two primary workflows: local state modifications and peer-to-peer synchronization.

3.4.1 Local State Modification

When a warehouse operator executes a command, the CLI acts as a gRPC client, sending a synchronous request to its local host node over `localhost`. Because the system prioritizes availability, the local node processes the transaction, ticks its own vector clock, and commits the data to its embedded SQLite database immediately, without waiting for consensus from the broader network.

3.4.2 Peer-to-Peer Synchronization

To achieve eventual consistency, peer nodes interact via a background gossip protocol. Periodically, each node packages its current inventory ledger and vector clock into a gRPC `SyncLedger` request and broadcasts it to its known peers.

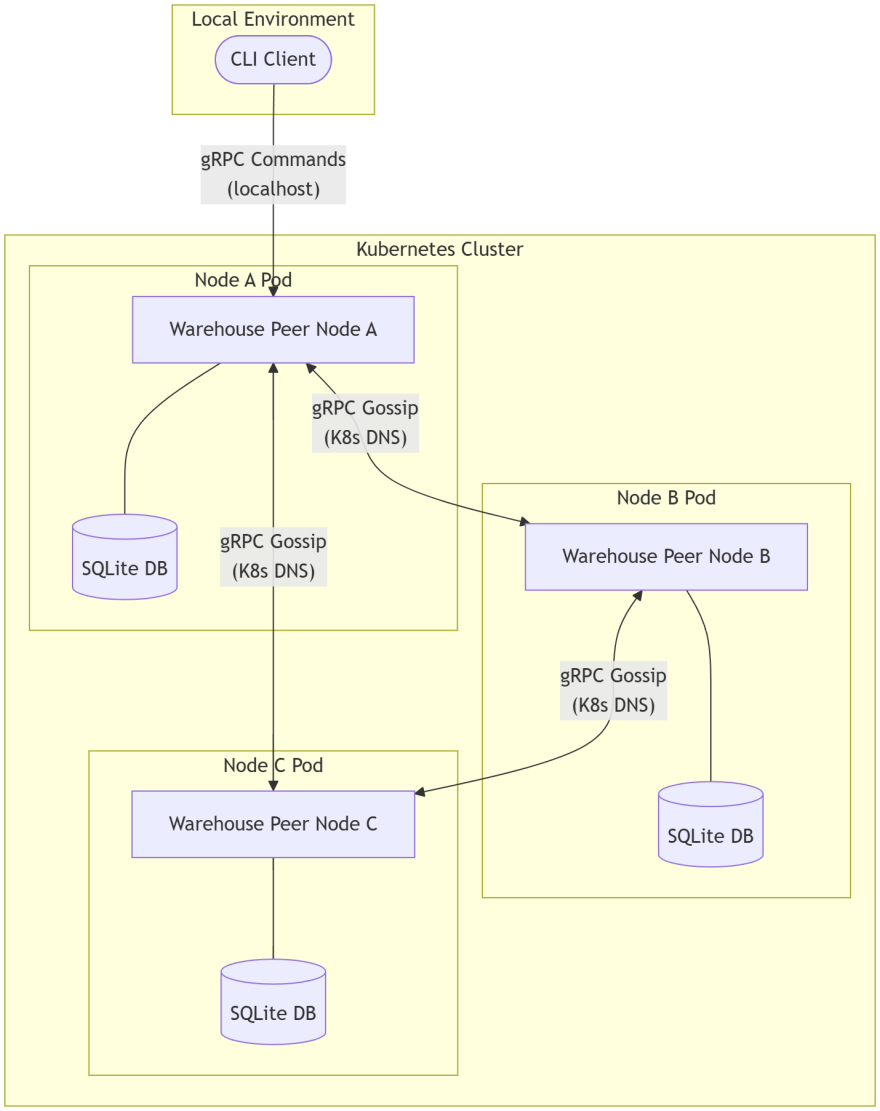

Figure 1: Component diagram of the system.

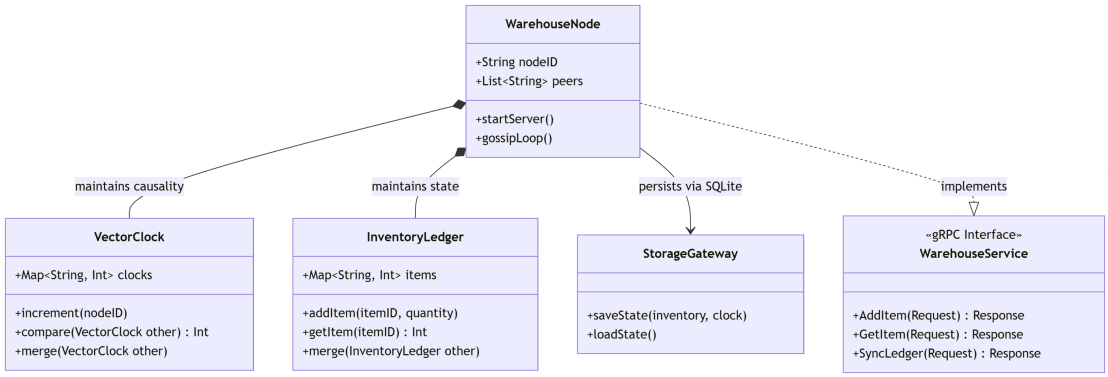

Figure 2: Class diagram showing the domain entities and their mapping to the SQLite persistence layer.

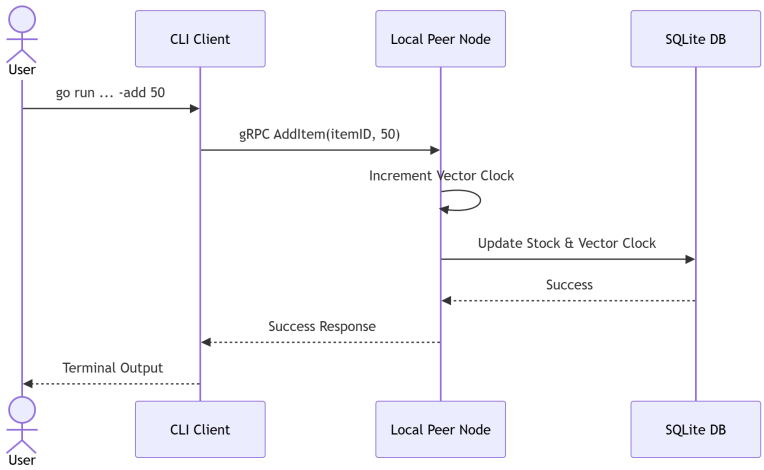

Figure 3: Sequence diagram illustrating the local write path from the user CLI to the node's database.

When a receiving node processes this payload, it compares the incoming vector clock against its own. If the clocks indicate that the updates happened concurrently (e.g., during a network partition), the receiving node executes the additive reconciliation protocol to mathematically merge the states before persisting the resolved ledger to disk.

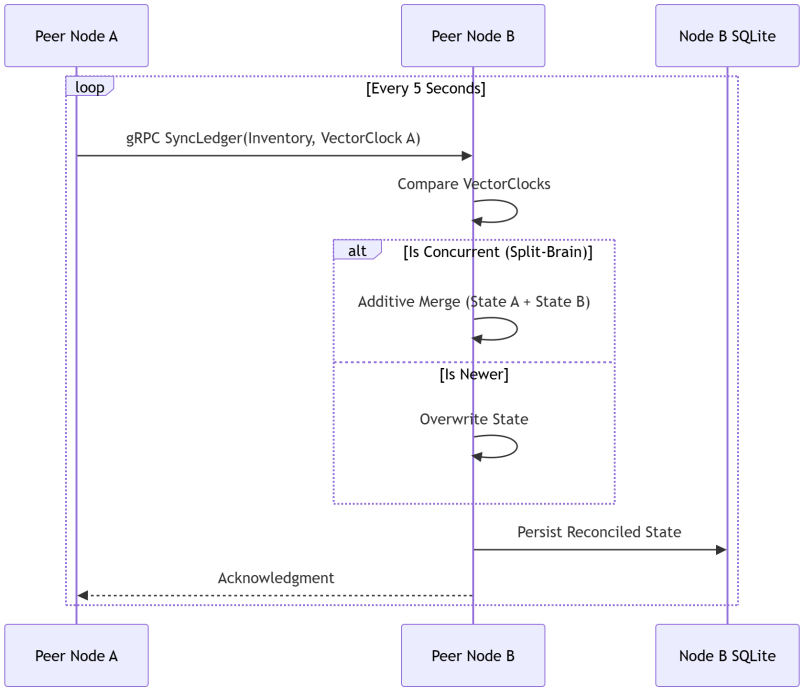

Figure 4: Sequence diagram detailing the background gossip protocol and state reconciliation.

3.5 Behaviour

The system's components exhibit distinct behaviours regarding state management and event handling.

Component Statefulness

- **CLI Client:** The client interface maintains no internal state. It behaves purely as a transient command executor, waking up to serialize a user input into a gRPC request, printing the node's response to the terminal, and immediately terminating.
- **Warehouse Peer Nodes:** The nodes are highly stateful daemon processes. They continuously maintain the global inventory ledger and causality history (Vector Clocks) in memory, serving as the definitive source of truth for their local warehouse.

State Updates The Warehouse Peer Nodes are exclusively in charge of updating the

system's state. Because there is no master database, state mutations occur independently on each node under two specific conditions:

1. **In Response to Local Commands:** When a node receives a write operation from its local CLI, it immediately ticks its vector clock forward to establish a new logical timeline and mutates its local inventory. It then persists these changes to the SQLite database.

2. **In Response to Gossip Events:** When a node receives a background sync message from a peer, it evaluates the incoming vector clock. If the incoming state is strictly newer, the node overwrites its local state. If the vector clocks indicate a *split-brain* scenario (concurrent updates during a partition), the node triggers a reconciliation behaviour, mathematically adding the divergent inventory deltas together to form a new, resolved state before persisting it.

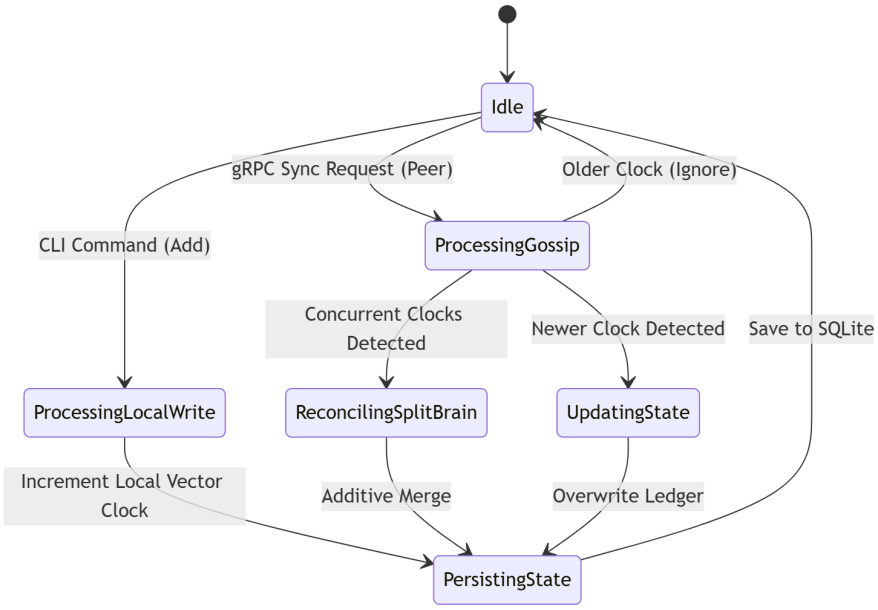

Figure 5: State diagram depicting how a Warehouse Node processes local writes and reconciles concurrent peer updates.

3.6 Data and Consistency Issues

Persistent Data Storage

The system requires persistent storage for two critical pieces of data: the **Inventory Ledger** (mapping item IDs to stock quantities) and the **Vector Clocks** (mapping node IDs to logical timestamps). This data is stored locally on the disk of each Warehouse Peer Node. Maintaining a local, durable copy of the state is essential to fulfill the

Availability and Partition Tolerance requirements, allowing local warehouse operations to proceed without blocking even if the WAN connection drops.

Storage Strategy

The persistent data is stored using an embedded **SQLite** database. While SQLite is fundamentally a relational database, the application logic interacts with it largely as a lightweight Key-Value store (Item $\rightarrow$ Quantity). This embedded, serverless approach was chosen because it requires no external database cluster to maintain, completely eliminates the single point of failure inherent to centralized databases, and allows a node to perfectly recover its state following a crash or container restart.

Database Queries and Concurrency

Only the Warehouse Peer Node directly queries its local database. These queries are executed under two primary circumstances:

- **Local User Commands:** The node queries and updates the database synchronously when a user submits a read or write request via the local CLI.
- **Background Network Syncs:** The node queries the database to package its state for gossip broadcasts and writes to the database when merging incoming states from peers.

Because local user interactions and background network syncs happen asynchronously, the node frequently handles concurrent reads and writes. This concurrency is safely managed by combining Go's native synchronization primitives with SQLite's internal file-locking, preventing race conditions when the node is actively reconciling a split-brain merge.

Shared Data and Consistency Model

To achieve eventual consistency across the distributed environment, peer nodes must continuously share their complete Inventory Ledgers and Vector Clocks with one another. This data is transmitted as serialized gRPC payloads during periodic gossip cycles. By sharing both the physical inventory quantities and the causal timeline (vector clocks), receiving nodes can accurately identify whether updates happened sequentially or simultaneously, triggering the additive reconciliation protocol only when a true network partition has occurred.

3.7 Fault-Tolerance

Data Replication

The system implements full, multi-master data replication across all peer nodes in the cluster. To completely eliminate single points of failure and ensure uninterrupted local warehouse operations, every node maintains a complete, eventual copy of the global inventory ledger. This replication is executed continuously via a background gossip protocol, where each node periodically broadcasts its full state (the inventory quantities and vector clocks) to its known peers.

Timeouts and Retry Mechanisms

Rather than implementing a rigid, synchronous heart-beating mechanism, the system

relies on asynchronous gRPC timeouts and periodic broadcast loops. When a node attempts to sync with a peer that is offline or unreachable due to a network partition, the gRPC request safely times out, preventing the active node's execution threads from blocking. The continuous nature of the gossip protocol acts as an implicit, infinite retry mechanism: if a node fails to reach a peer during one cycle, it simply attempts to transmit its updated state again during the next scheduled tick.

Error Handling and Component Failure

Because the architecture prioritizes Availability and Partition Tolerance, the system handles component failures gracefully by isolating them:

- **Node Crashes:** If a Warehouse Peer Node suffers a hardware or process failure, the rest of the decentralized cluster is entirely unaffected and continues to process local read and write operations. When the failed node reboots, it loads its last known state from its local SQLite database and immediately resumes receiving gossip payloads, allowing its peers to seamlessly push the missed data and bring the recovered node back up to speed.

- **Network Partitions:** If communication between nodes is severed, the system handles the error by allowing nodes to diverge into a split-brain state, prioritizing local uptime over global consistency. Once the network heals, the vector clocks accurately detect the concurrency error, and the system mathematically merges the divergent state histories to guarantee zero data loss.

3.8 Availability

Caching Mechanisms

The system does not implement a dedicated external caching layer. Instead, the decentralized architecture inherently acts as a caching mechanism. Because every peer node maintains a complete, persistent replica of the global inventory in its local SQLite database, read queries initiated by the CLI are served instantly from the local disk without incurring network round-trip latency.

Load Balancing

There are no central load balancers deployed within this infrastructure. The architecture avoids centralized bottlenecks entirely. Each warehouse operator interacts exclusively with their specific local peer node via the CLI, naturally distributing the user load geographically. Inter-node communication relies on direct peer-to-peer gRPC calls routed through Kubernetes internal DNS, rather than funneling traffic through a central proxy or load balancer.

Network Partitioning Behaviour

In the event of a network partition, the system explicitly prioritizes Availability over strict Consistency.

- This is due to warehouse operations not being able to halt simply because an external WAN connection fails; users must be able to continue logging local stock additions and deductions.

- When isolated, a node continues to process local read and write commands, deliberately diverging from the global state. It continues to tick its local vector clock to track the causal history of its isolated operations. Once network connectivity is restored, the background gossip protocol resumes. The nodes exchange their divergent vector clocks, detect the concurrent modifications, and automatically execute the additive reconciliation protocol to mathematically merge the inventory deltas into a unified global state without data loss.

3.9 Security

In its current prototype phase, the system operates within a trusted local network and does not implement active security measures. However, migrating this architecture to a production WAN environment requires a comprehensive zero-trust security model. Below is the designed security architecture planned for future iterations.

Authentication

Currently, the system lacks authentication. In production, authentication must be implemented at two distinct layers:

- **Peer-to-Peer:** Mutual TLS (mTLS) will be required for all gRPC communication between warehouse nodes. This ensures that only cryptographically verified nodes can join the cluster and broadcast gossip payloads, preventing rogue nodes from injecting malicious inventory states.
- **Client-to-Node:** Operators will authenticate via a centralized Identity Provider to receive a short-lived JSON Web Token (JWT). The CLI will attach this JWT to the metadata of every gRPC command sent to the local node.

Authorization

Currently, any user with CLI access can modify inventory. Future iterations will implement **Role-Based Access Control** by inspecting the claims within the user's JWT.

- **Warehouse Operators:** Granted restricted access to `AddItem` and `GetItem` methods for their specific physical location.
- **Global Auditors:** Granted read-only access to query inventory and vector clock states across the entire cluster.
- **System Administrators:** Granted elevated privileges to force ledger resets or manually remove failing nodes.

Cryptographic Schemas

The current prototype transmits gRPC payloads in plaintext and stores SQLite data unencrypted. A production deployment will enforce:

- **Data in Transit:** All gRPC channels will be secured using TLS 1.3, preventing packet sniffing and man-in-the-middle attacks over the public internet.

- **Data at Rest:** The embedded SQLite databases on each node will utilize SQLCipher to provide transparent 256-bit AES encryption of the local ledger files, ensuring that a compromised container or stolen hard drive does not expose enterprise inventory data.

4 Implementation

This section details the specific technology stack and protocols utilized to implement the distributed peer-to-peer architecture.

- **Network Protocols:** The system relies exclusively on **gRPC** for all communication between the CLI client and the local nodes, as well as for the peer-to-peer gossip protocol. gRPC operates over HTTP/2 and TCP, providing a robust, low-latency communication channel that handles multiple concurrent requests.
- **Data Representation:** In-transit data is serialized using **Protocol Buffers (protobuf)**. Unlike text-based formats like JSON or XML, protobuf provides a highly compressed binary format. It enforces strict type contracts between distributed components, ensuring that an `AddItem` payload or a `SyncLedger` state transfer cannot be malformed during transmission.
- **Database Queries:** The local inventory ledger and vector clocks are persisted using SQLite. The components query this embedded database using standard **SQL** statements (e.g., `INSERT`, `UPDATE`, `SELECT`) via Go's native `database/sql` package.
- **Authentication and Authorization:** As designed in the Security section, the current prototype bypasses active access control to focus on core distributed systems principles. A production deployment will authenticate components via **JSON Web Tokens** and authorize users using **Role-Based Access Control**, mapping token claims to specific gRPC method permissions.

4.1 Technological details

The following frameworks and technologies form the foundation of the project:

- **Go:** The entire codebase (CLI and Warehouse Nodes) is written in Go. Go was selected for its native gRPC support and its powerful concurrency primitives. Goroutines and channels are used extensively to manage the non-blocking background gossip protocol while simultaneously serving local CLI requests.
- **SQLite and go-sqlite3:** Used for serverless, embedded data persistence. It eliminates the need for an external database daemon, perfectly serving this decentralized architecture.
- **Docker:** Used to containerize the compiled Go application, ensuring environmental consistency across different deployment targets.

- **Kubernetes (Minikube):** Acts as the orchestration layer to simulate a geographically distributed Wide Area Network on a local machine.
- **Project Calico:** Integrated as the Container Network Interface (CNI) for the Minikube cluster. Calico was specifically required to enforce Kubernetes `NetworkPolicy` rules, allowing the system to accurately simulate packet-dropping network partitions without destroying the underlying pod infrastructure.

5 Validation

5.1 Automatic Testing

Unit Testing
Individual components were unit-tested using Go's native `testing` package. To isolate the application logic from the file system, all database unit tests utilized an ephemeral, in-memory SQLite instance (`:memory:`).

- **Vector Clocks (`vclock_test.go`):** Tested the deterministic logic of causality tracking, ensuring the system accurately calculates Equal, Before, After, and Concurrent relations. Merge operations were validated to ensure the resulting clock always contains the maximum integer value for each node.
- **Storage (`storage_test.go`):** Verified that stock quantities and vector clocks are correctly serialized and persisted to the SQLite layer, and that data retrieval perfectly reconstructs the domain entities.
- **Server API (`server_test.go`):** Validated the gRPC endpoints. Specifically, `TestSyncLedger_ConcurrentMerge` simulated a split-brain scenario by passing divergent vector clocks and verifying that the server correctly performed an additive merge (e.g., adding 50 and 100 to yield 150).
- **Gossip Protocol (`gossip_test.go`):** Verified that the background daemon successfully builds synchronization payloads. `TestSendGossipToOfflinePeer` ensured that network connection errors trigger graceful failures rather than system crashes.

End-to-End Testing
Because the system's core complexity lies in its distributed network topology, integration and E2E testing were performed in a production-like environment using automated Bash scripts against a local Kubernetes cluster (Minikube).

- **Normal Execution (`01-normal-execution.sh`):** Injects data into a single node and verifies that the background gossip protocol eventually propagates the state to all peers. Tests the core Availability requirement.

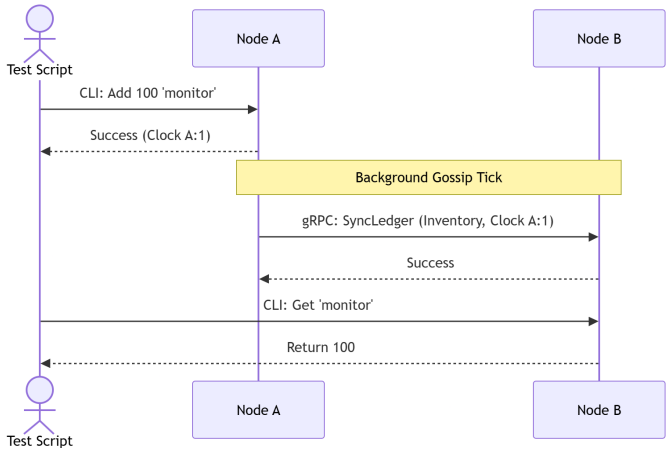

Figure 6: Sequence diagram illustrating standard state propagation via the gossip protocol.

- **Node Failure** (`02-node-failure.sh`)**:** Tests crash fault tolerance by scaling a node's pod to zero replicas while actively injecting data into the surviving cluster. Verifies that the crashed node completely recovers its missed ledger upon reboot.

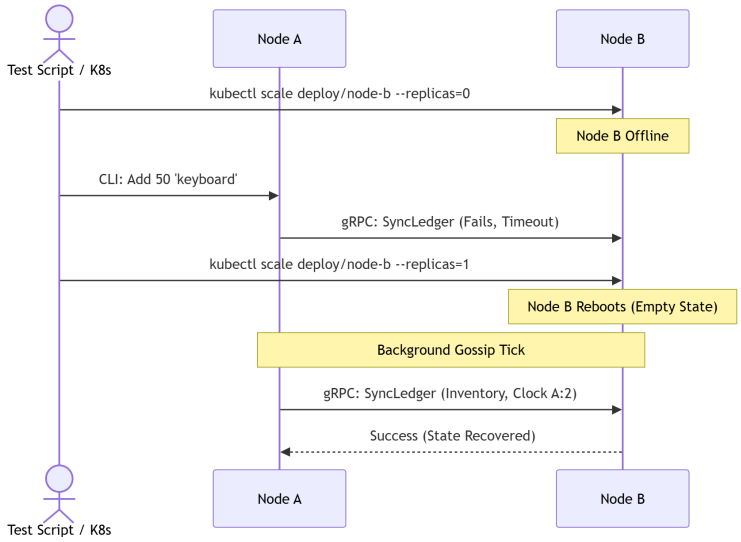

Figure 7: Sequence diagram detailing a simulated container crash and subsequent automated state recovery.

- **Network Partition** (`03-network-partition.sh`)**:** Tests Vector Clock causality and split-brain reconciliation. A Calico `NetworkPolicy` isolates a node to simulate a WAN partition. Conflicting deductions are injected into the split network. Once the firewall is deleted, the script verifies that the cluster mathematically resolves the overlapping states without data loss.

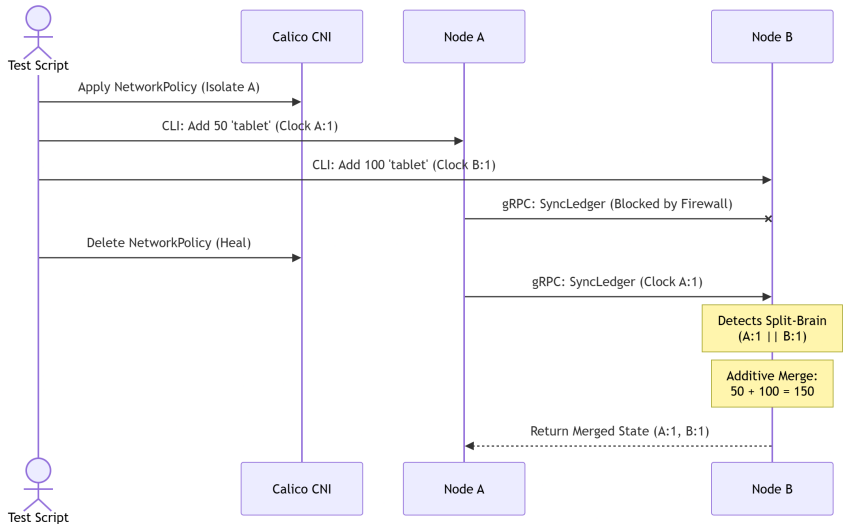

Figure 8: Sequence diagram depicting a Layer-3 network partition and the resulting vector clock state reconciliation.

5.2 Acceptance test

While edge-case failures and state reconciliation were validated automatically, manual acceptance testing was performed to evaluate the system's non-functional Usability and Performance requirements.

- **What was tested:** A human operator manually interacted with the CLI to submit various `-add`, `-get`, and `-all` commands against local port-forwarded pods, observing the terminal output.
- **Why it wasn't automatic:** The clarity of the terminal UI, the readability of error messages (e.g., submitting invalid string quantities), and the perceived responsiveness of local commands (verifying that the execution happens in under 100 milliseconds) are inherently subjective UX factors that require manual validation.

6 Deployment

This section details the infrastructure engineering, containerization strategies, and orchestration mechanisms required to deploy the distributed warehouse inventory system from scratch. The deployment architecture leverages local virtualization to accurately simulate a geographically distributed WAN environment.

6.1 Prerequisites and Environment Setup

To successfully compile and execute the system, the host machine must be provisioned with a specific toolchain. The project relies on the following software dependencies:

- **Go:** Required to compile the underlying application source code and resolve external gRPC and SQLite module dependencies.
- **Docker Engine:** Utilized as the primary container runtime to build and isolate the application environments.
- **Minikube:** Acts as the local Kubernetes engine, provisioning the virtualized node infrastructure required to simulate the cluster topology.
- **Kubectl:** The Kubernetes command-line tool required to apply declarative manifests and interact with the cluster control plane.

6.2 Containerization and Build Engineering

The application is strictly containerized to guarantee environmental consistency across the development, testing, and deployment phases. The containerization process is engineered using a multi-stage Docker build strategy defined within a `Dockerfile`.

The first stage utilizes the official `golang:1.21-alpine` image to compile the binaries. Because the system relies on an embedded SQLite database, the Go compiler requires CGO to be enabled (`CGO_ENABLED=1`), necessitating the installation of `gcc` and `musl-dev` within the build container. The final stage copies the compiled binaries into a minimal Alpine Linux base image. This multi-stage approach reduces the final attack surface and the container's memory footprint, ensuring the warehouse nodes remain lightweight enough to scale efficiently.

6.3 Cluster Orchestration and Networking

The system's distributed nature is orchestrated using Kubernetes via Minikube. A standard Kubernetes cluster is insufficient for this project's testing requirements because the default Minikube Container Network Interface does not support or enforce `NetworkPolicy` objects by design.

To simulate the network partitions necessary for validating the AP architecture, the cluster must be booted with Project Calico. Calico acts as an advanced networking and network policy provider. It provides the necessary engine to enforce routing rules at the OSI layer 3 and 4. By appending the `--cni=calico` flag during the cluster initialization, Minikube automatically provisions the Calico DaemonSets, allowing the system to selectively drop packets and isolate specific warehouse nodes to test vector clock reconciliation.

6.4 Deployment Manifests Architecture

The Kubernetes topology is constructed using declarative YAML manifests. The architecture is composed of the following core infrastructural objects:

- **Deployments:** Each warehouse peer (Node A, Node B, Node C) is governed by its own independent Kubernetes Deployment object. These deployments manage the

underlying Pod replica sets, ensuring that if a container crashes, the orchestrator automatically restarts it to maintain high availability.

- **Persistent Storage:** To persist the SQLite database across unexpected container crashes within the same pod lifecycle, the deployments map an `emptyDir` volume to the container's storage path. While a production environment would utilize persistent `StatefulSets` and external `PersistentVolumeClaims` (PVCs), `emptyDir` is sufficient for local session-based testing.
- **ClusterIP Services:** Kubernetes `Service` objects are deployed to provide stable internal DNS resolution. Because Pod IP addresses change upon restart, the services ensure that nodes can consistently discover and gossip with one another using static hostnames (e.g., `node-a-svc`).
- **Network Policies:** Isolated YAML files define Calico `NetworkPolicy` specifications. When applied, these objects dynamically modify the cluster's ingress and egress rules to sever communication to specific pods, intentionally inducing a split-brain scenario.

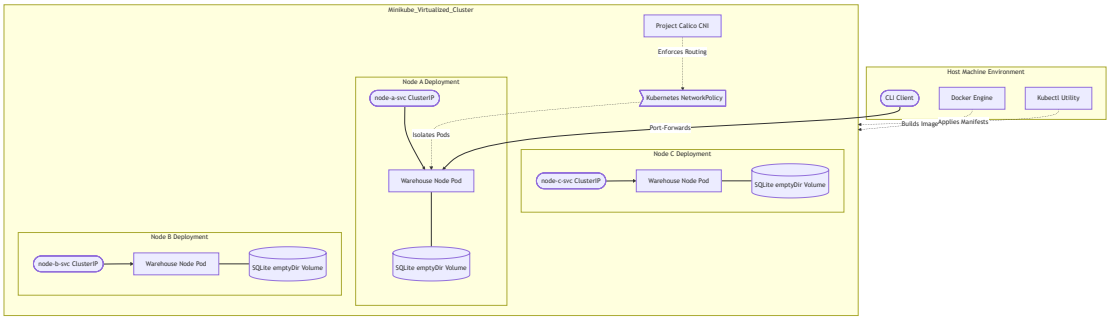

Figure 9: Deployment diagram illustrating the Kubernetes orchestration, Calico CNI integration, and Pod-to-Service mappings.

6.5 Execution and Verification Guide

To deploy the software from scratch, a user must execute the following sequential instructions from the root directory of the project workspace:

1. Bootstrap the Orchestration Cluster

Initialize the Minikube environment and specify Calico as the underlying networking plugin:

```
minikube start --cni=calico
```

2. Build and Load the Container Image

Compile the multi-stage Docker image and load it directly into Minikube's internal container registry to bypass the need for an external cloud repository:

```
docker build -t warehouse-node:latest .
minikube image load warehouse-node:latest
```

3. Apply the Infrastructure Manifests
Instruct Kubernetes to parse the declarative YAML files and provision the Deployments and Services:

```
kubectl apply -f k8s/
```

Expected Outcomes:
Following the execution of the `kubectl apply` command, the user can run `kubectl get pods`. The expected outcome is three distinct pods (representing the three warehouse nodes) transitioning from the `ContainerCreating` phase to a `Running` status. Once all pods report `1/1 READY`, the decentralized gRPC network is fully operational and passively listening for CLI interactions.

7 User Guide

The distributed warehouse system is operated entirely via a Command-Line Interface (CLI). This design abstracts the complex gRPC networking and background gossip protocols away from the end-user, providing a simple, synchronous terminal experience.

Below are the step-by-step instructions to perform standard warehouse operations.

7.1 Connecting to a Node

Before executing inventory commands, the operator must establish a connection to their local warehouse node. In this Minikube deployment, this is achieved by opening a terminal and port-forwarding the Kubernetes service to the local machine.

Command:

```
kubectl port-forward svc/node-a-svc 50051:50051
```

Expected Outcome: The terminal will block and display `Forwarding from 127.0.0.1:50051 -> 50051`, indicating the CLI can now communicate with Node A.

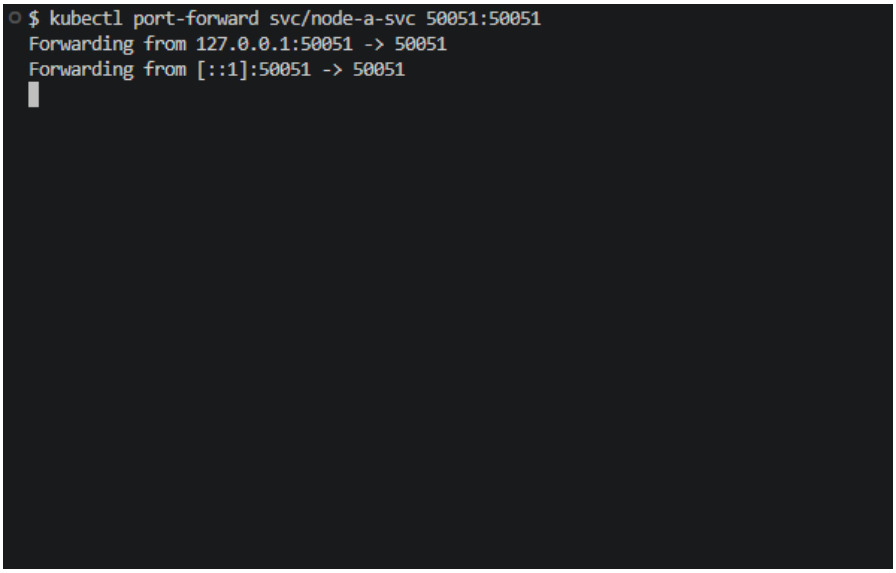

Figure 10: Terminal output confirming a successful port-forward connection to Node A.

7.2 Adding Inventory (Incoming Shipments)

When a new shipment arrives at the warehouse, the operator uses the `-add` flag to increment the local stock.

Command:

```
go run ./cmd/cli -target localhost:50051 -item monitor -add 50
```

Expected Outcome: The CLI sends the gRPC request, the node updates its SQLite database, and the terminal prints a success message along with the node's updated vector clock showing a new logical tick.

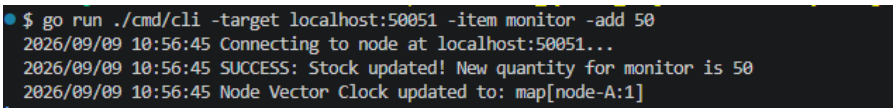

Figure 11: CLI output demonstrating a successful addition of 50 monitors.

7.3 Deducting Inventory (Local Sales)

When items are sold or shipped out, the operator deducts stock by passing a negative integer to the `-add` flag.

Command:

```
go run ./cmd/cli -target localhost:50051 -item monitor -add -20
```

Expected Outcome: The terminal confirms the stock reduction and displays the new total quantity alongside the incremented vector clock.

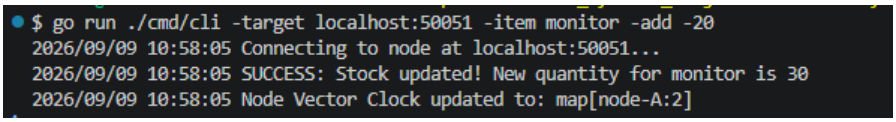

Figure 12: CLI output showing the successful deduction of 20 monitors.

7.4 Querying Global State

An operator can audit the complete inventory ledger stored on their local node using the `-all` flag. Because of the background gossip protocol, this local read will eventually reflect the global state of the entire multi-warehouse cluster.

Command:

```
go run ./cmd/cli -target localhost:50051 -all
```

Expected Outcome: The terminal outputs a complete map of all inventory items and their current quantities, followed by the node's current understanding of the global vector clock.

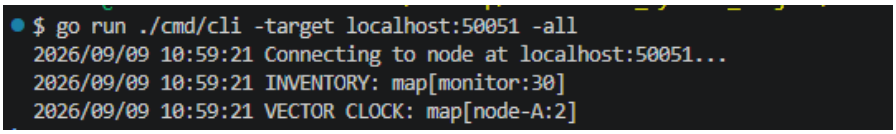

Figure 13: The CLI returning the complete, synchronized inventory ledger from the local node.

7.5 Executing Automated Scenarios

To demonstrate distributed systems edge cases (such as network partitions and crash recoveries), users can execute the provided Bash scripts located in the `demo/` directory.

Command:

```
./demo/03-network-partition.sh
```

Expected Outcome: The script will automate the creation of a Calico NetworkPolicy, inject conflicting split-brain data into isolated nodes, heal the network, and print the mathematically reconciled inventory state to prove the eventual consistency guarantees.

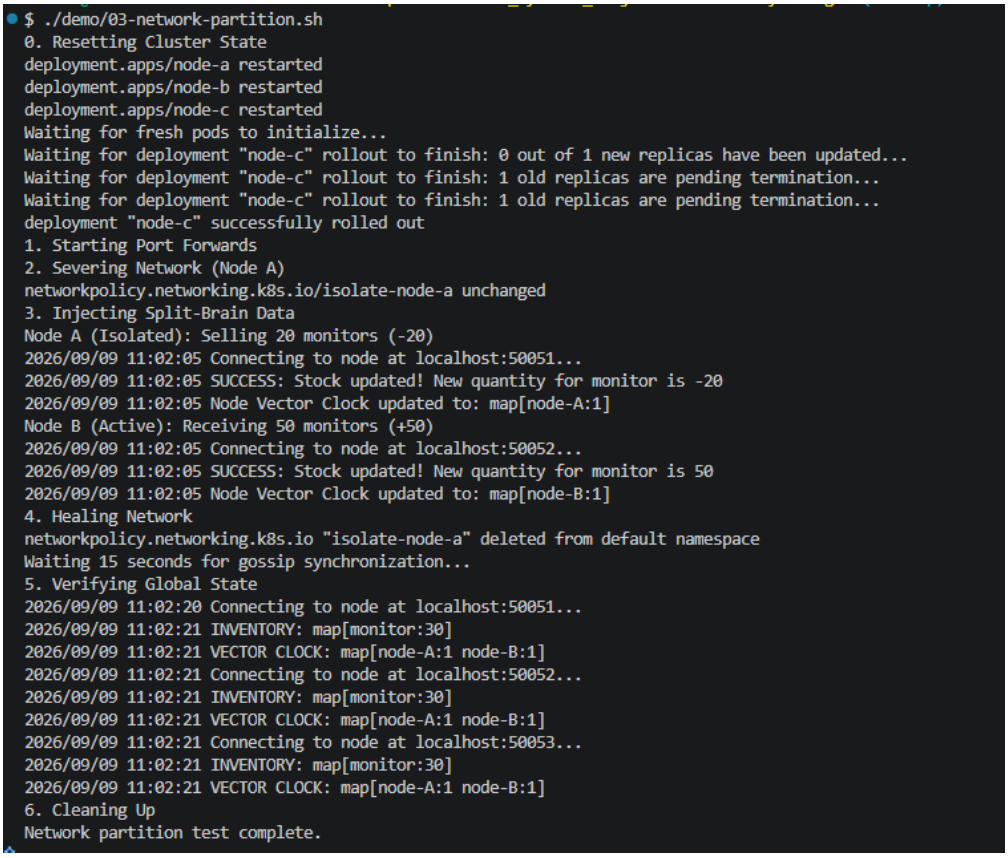

Figure 14: Terminal output from the automated network partition and state reconciliation script.

8 Self-evaluation

As the sole maintainer of this project, I was responsible for the entire software lifecycle. This included requirements elicitation, system modeling, backend implementation in Go, and the infrastructure orchestration using Docker and Kubernetes. Operating independently required balancing theoretical distributed systems concepts with practical DevOps engineering, particularly when researching and integrating Project Calico to simulate realistic network partitions.

Strengths of the Product

- **Resilient AP Architecture:** The system successfully prioritizes Availability and Partition Tolerance. By eliminating the central database layer, local warehouse operations are never blocked by upstream WAN outages.
- **Deterministic State Reconciliation:** The implementation of Vector Clocks combined with an additive merge protocol mathematically guarantees eventual

consistency. The automated Bash tests prove that the system handles concurrent split-brain deductions without dropping data.

- **Advanced Infrastructure Validation:** Utilizing Minikube with the Calico CNI provided a highly realistic testing environment. Severing communication at OSI Layer 3 via Kubernetes `NetworkPolicy` offers a much stronger architectural validation than simply mocking network errors within the Go codebase.
- **Lightweight Persistence:** The use of embedded SQLite ensures durable, serverless state recovery upon node crashes while maintaining an incredibly minimal memory footprint for the containers.

Weaknesses of the Product

- **Naive Gossip Payload Sizing:** Currently, nodes broadcast their entire inventory ledger during every sync cycle. While this works perfectly for a prototype cluster, transmitting the full database state consumes excessive bandwidth and will not scale efficiently for a massive enterprise catalog.
- **Static Peer Discovery:** The system relies on hardcoded Kubernetes Service DNS names to locate peers. It lacks a dynamic discovery registry to automatically detect new warehouse nodes joining the WAN.
- **Security Implementation:** Due to the potential complexity of stabilizing the state reconciliation math, strict security features (mTLS, JWT authentication, SQLCipher encryption) were designed theoretically but omitted from the active container builds.

9 Future works

While the current implementation fulfills the core requirements for an AP-oriented, eventually consistent distributed ledger, several optimizations and unimplemented features are planned for future iterations.

Optimized Delta Synchronization
Currently, nodes broadcast their entire SQLite inventory during every gossip cycle. As the enterprise catalog grows, this full-state transmission will consume excessive network bandwidth. Future iterations would implement delta-based synchronization. By evaluating vector clocks before transmitting the payload, a node would only serialize and send the specific item quantities that have mutated since the receiving node's last known tick.

Dynamic Peer Discovery
The prototype relies on static Kubernetes `Service` manifests to locate peers (e.g.,

`node-a-svc`). To support elastic WAN scaling where new physical warehouses are added or removed dynamically, a dedicated service discovery registry would be introduced. Nodes would register themselves upon container initialization and query the registry to dynamically build and update their active peer gossip lists.

Comprehensive Security Integration
As designed theoretically in the Security section, the system currently lacks active access control and encryption. Bringing the application to production readiness would involve:

- **Transport Security:** Configuring the gRPC server and client dialers to use Mutual TLS (mTLS). This requires provisioning certificates for each node via a local Certificate Authority (CA) or integrating a service mesh.
- **Access Control:** Integrating a gRPC unary interceptor in the Go backend to parse JSON Web Tokens from the incoming request metadata, strictly enforcing Role-Based Access Control prior to executing any SQLite transactions.
- **Data at Rest:** Swapping the standard `go-sqlite3` driver for `sqlcipher` to provide transparent AES-256 encryption on the local disk, injecting the necessary decryption keys into the pods via Kubernetes Secrets.

Physical Constraint Alerts
Additive state reconciliation mathematically resolves data conflicts, but it can result in negative inventory if isolated nodes concurrently sell overlapping stock. Future work would introduce an asynchronous alerting service (e.g., publishing to an Apache Kafka topic) whenever a reconciliation merge results in a sub-zero quantity, immediately notifying human managers to initiate physical stock transfers or cancel local orders.