Distributed Inventory System

Project Distributed Systems

for the

Master of Science

from the Course of Studies Computer Science

at the University of Stuttgart

by

Luca Schwarz, Johannes Hartmann, Patrick Baisch

10.02.2024

Due Date	16.10.2023-08.02.2024
Student ID	3731224, 3743267, 3125018
Reviewer	Marco Aiello

Author’s declaration

Hereby I solemnly declare:

1. that this Project Distributed Systems, titled *Distributed Inventory System* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Project Distributed Systems has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Project Distributed Systems in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Stuttgart, 10.02.2024

Luca Schwarz, Johannes Hartmann, Patrick Baisch

Abstract

Contents

Acronyms

List of Figures

List of Tables

Quellcodeverzeichnis

1 Introduction

As part of the course project an inventory system shall be developed. The system will be a client server architecture where the servers are responsible to keep the data and process updates regarding goods and stock information. The clients can request the currently available goods and the amount and send update information if new goods are available or if goods are taken out of stock. For the updates it's important to ensure strong consistency and ordering of events such that all clients have the current information. Within the servers it shall be possible to add additional nodes to the system dynamically and to handle different failure cases while still being able to process requests.

2 Project Requirement Analysis

2.1 Architectural Description

Client-Server Architecture: Our system will adopt a client-server model, which consists of multiple clients interfacing with a cluster of server nodes. The clients are responsible for querying current stock levels and submitting updates for processing. The server nodes serve these stock level requests and manage inventory updates. Amongst the server nodes, a leader will be elected to coordinate updates and ensure consistency across the distributed system.

2.2 Dynamic Discovery of Hosts

Server-side: Upon initiation, each server node will broadcast its presence and listen for existing members of the system to construct a current view of the cluster. The broadcast message from the joining server is only responded by the leader of the system. The message contains an array with all ip addresses currently active within the system, as well as the information which node is the leader.

When the servers are started, no leader is chosen at this point of time. In this case, the first node which does not get a reply of his broadcast declares himself as leader and answers the requests of the other nodes. To ensure, that only one leader is chosen at the starting point, a random sleep is implemented, which ensures the existence of only one leader.

This dynamic discovery protocol allows the system to scale horizontally without manual configuration.

Client-side: Clients are designed to automatically detect server nodes in the system. As well as the server-side implementation a broadcast message is sent, but not with the intention to join the cluster. In the Client-side case the ip address is not added to the server node pool. The reply to the broadcast is done by the leader, sending an array of all members to the client. The client uses one randomly chosen address to communicate with. If the reply of an request by an client is not answered by the node, another discovery will be done to get an updatet version of the list with nodes.

This enables seamless interaction with the inventory system, ensuring that clients can always locate a server node to process their requests.

2.3 Fault Tolerance

Our system is engineered to handle different types of failures, ensuring continuous operation: Leader Failure: If the current leader server fails, the remaining servers will initiate a leader election to select a new leader. In the meantime no operation is permitted, because our system uses strong consistency. As soon as a new leader is elected, the system returns to normal process. This ensures that the system become available after a crash of the leader.

Server Crash: In the event of a server crash, client requests are automatically redirected to other operational server nodes. A health check mechanism and a server list update protocol ensure that clients and servers are aware of the available nodes in real-time. When a node does not send a Heartbeat, it will be removed from the list by every server, including the leader, so the normal processing can continue.

Retry Logic: To handle transient failures, the system will implement a retry mechanism. This ensures that, for instance, during a purchase processing, if a write operation fails, the system will retry the operation. We will have a limit to prevent excessive retries that could overwhelm the system, allowing it to fail and recover.

2.4 Leader Election

Consensus on Updates: Write operations, critical for inventory synchronisation, are managed by the leader node. Also new nodes which want to join the group exchange information with the leader. The election Algorithm will be the bully Algorithm. When an election is triggered (the heartbeat of the leader is missing), the first Server, which realises it, starts an election, using an ID as information multicastet to every other server. If another Server has an higher id, he claims to become the leader, “bullying” the server with a lower ID and denying his election. This will continue unitl the server with the highest ID wins the election. He then sends a message, declaring himself as new leader and the workflow can continue.

2.5 Ordered Reliable Multicast

Our system uses FIFO ordered reliable multicast to ensure inventory updates, like purchases or deletions, are processed sequentially, maintaining data consistency and preventing access to outdated item statuses.

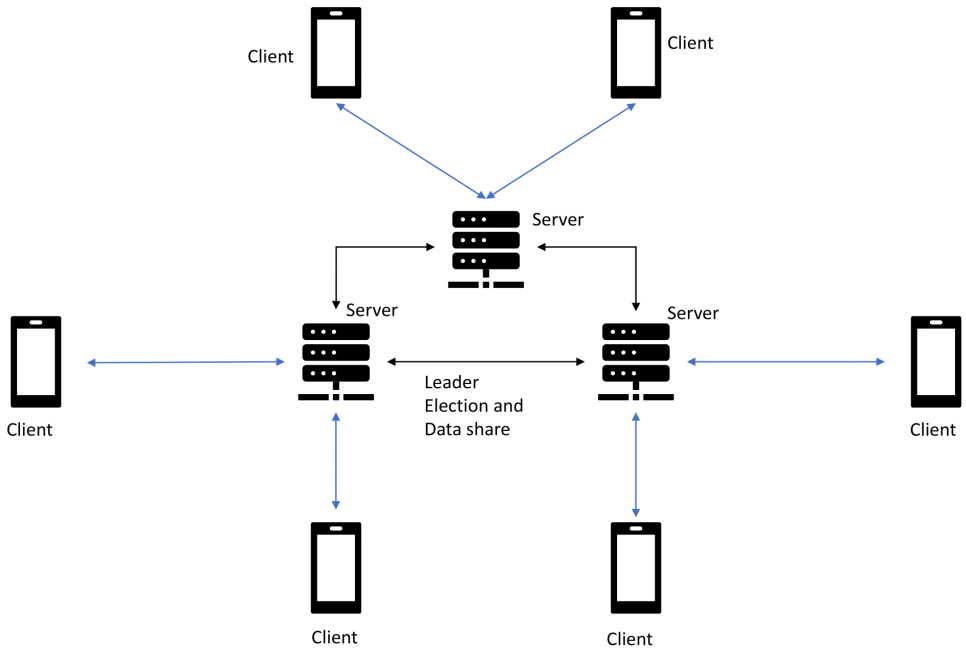

Figure 2.1: Architecture Diagram

3 Component Description

4 Node

Each node in the system will check on startup if there are other nodes already running within the network. This is done via a broadcast message containing the nodes ID and IP Address.

4.1 Leader

We use the bully algorithm to determine a uniqe leader out of all nodes. The bully algorithm ensures that there is always a leader.

The leader mostly behaves the same as all the other nodes. Additional tasks are maintaining the group-view and coordinating write operations.

The group view is maintained by listening on broadcasts for new nodes and regularly checking if nodes in the group-view are still alive. The leader will accept the new server and distribute the updated group-view to all nodes.

As consistency is of high importance for the inventory system write operations are coordinated by the leader to ensure that all nodes have up-to-date information.