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

Luca 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. **Luca** 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.

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 total ordered reliable multicast to ensure inventory updates, like purchases or sales, are processed sequentially and data consistency is maintained over all nodes. This also prevents that a host has outdated information.

In the ordered reliable multicast the leader acts as the sequencer. All business messages are sent to the leader, the leader then sends the messages to all hosts via udp multicast. Each message contains a sequence number and the actual content. When a host receives a message from the leader the message is put into a log and an acknowledge message is sent back to the leader. Upon receiveing an acknowledge message the leader adds the sender to the acknowledge list and checks if all hosts acknowledged the message. This is due to the fact that we strive for full consistency over all active hosts. After receiveing enough acknowledgements the leader sends a commit message to all hosts triggering that the message is processed.

For each received commit message the host checks if the sequence number is strictly one higher than the sequence number kept by the host. If that's the case the message is processed. If that's not the case the host missed messages. To ensure consistency and ordering of the messages it requests from the leader all messages starting from it's current sequence number up to the received sequence number. The leader then sends the messages in the order of the sequence numbers. The messages can then be consumed by the host.

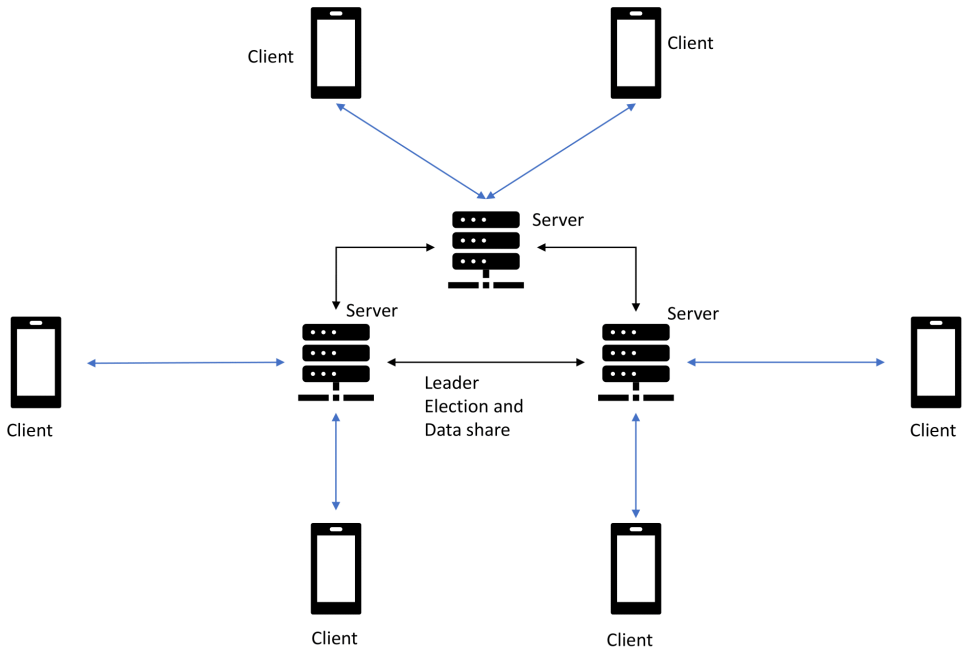

Figure 2.1: Architecture Diagram

3 ComponentDescription

3.1 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. If after a timeout no host responded the system assumes it's the first node and becomes leader. If other systems are already active the leader responds to the broadcast with the current list of hosts, it's own IP (the leader IP). The new host updates its own host list with the information received from the leader and sets the leader IP address.

Each node sends regularly heartbeats to all other nodes. It then checks if there are missed heartbeats from other nodes. If for a node the heartbeats are missed for to long it removes the host from the hosts list. If a node detects a failure of the leader a new election is started.

Each node listens for business requests from clients. If a business request is received by a host not being leader it forwards the request to the leader and waits for a response. Which is then forwarded to the requesting client.

3.2 Leader

The first leader is determined by the first host in the system. In case the leader fails a new leader is elected out of all nodes using the bully algorithm to determine a uniqe leader.

The leader behaves the same as all other nodes regarding heartbeat and processing of commited messages.

In addition to the above the leader responds to broadcast messages from new hosts and clients with the current hosts list.

Processes all business logic messages and distributes them using ordered reliable multicast to the other nodes. After a message is commited and processed the leader responds to the host which sent the message.

3.3 Client

The client serves a web frontend to interact with the system. When a user triggers a business request the client requests all currently active hosts and sends the request to one of the active hosts and waits for a response.

Patrick

4 Discussion

4.1 Fail Stop

Luca

4.2 Crash

Luca

4.3 Byzantine

Luca

4.4 Voting

Luca

4.5 Ordered Reliable Multicast

Johannes

4.6 Dynamic Discovery of Hosts

Luca

5 Summary