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

Contents

Acronyms

BFT	Byzantine Fault Tolerance
UUID	Universally Unique Identifier

List of Figures

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 as seen in figure 2.1, 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. Our main focus is the strong consistency, therefore if a new request is past from the client to a server, first the server reaches out to the leader, which will commit the change and pass it to all other servers. Writing requests are always passed to the leader, before they take place. Reading requests can be handled from a server node, the leader is not involved in this actions. When the leader dies, no bussiness logic can happen, except for reading, before a new leader is chosen.

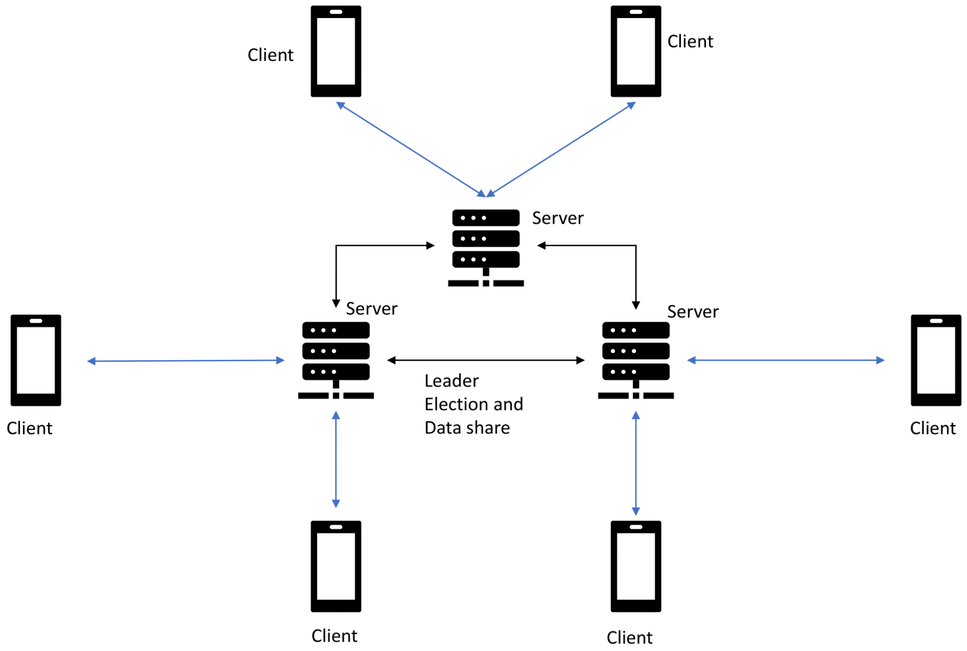

Figure 2.1: Architecture Diagram

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. As ID we use the IP Address as string to compare. 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.

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 Crash

As mentioned in section 2.3 our system is able to handle the crash of a server node, as well as the crash of the leader node. The procedure of the crashes differs, whether a node dies, or the leader dies.In case of a node dying, the system can operate as if nothing happens. Internally the other servers realize, that there is no heartbeat of the dead node, After 2 seconds without a heartbeat, the server node is taken out from the cluster and in the meantime the system operates as usual. The benefit of this approach is, that there is no downtime at all. But it can happen, that a request from a client is not registered and it has to be requested again, in order to handle the change.

A slightly different approach takes place, when the leader dies. In that case the system is not operational, until a new leader is elected. As soon as a node realizes the death of the leader, the election will be started and a new leader will be chosen. This ensures strong consistency, even without the leader but at the cost of a little downtime, where the requests from the clients will be paused, until the new leader is operational. Requests that are already placed but not finished, will be lost in order to keep the system consistent, therefore the user has to place the requests again.

4.2 Fail Stop

Fail Stop is the stop of a system, if a node/component is not operational. One of the primary advantages of fail-stop lies in its simplicity and predictability. When a component or node within a distributed system fails, it halts all operations immediately and unequivocally. This clear-cut behavior simplifies fault detection and recovery processes, as the failed component can be swiftly identified and isolated without ambiguity. Consequently, fail-stop facilitates efficient fault management strategies, minimizing downtime and enabling rapid system restoration.

However, fail-stop is not without its limitations and disadvantages. One notable drawback is the potential for abrupt service disruptions and loss of progress. Since fail-stop entails an immediate cessation of operations upon failure, ongoing tasks or transactions may be abruptly terminated, leading to potential data loss or service interruptions. This characteristic can be disruptive, particularly in scenarios where the system's continuity and uninterrupted operation are paramount.

In order to fullfill our strong consistency, we did not implement an fail stop procedure. But with the crash handling as described in 4.1, the other nodes will notice a failure and take over the ongoing operations.

4.3 Byzantine

Byzantine Fault Tolerance (BFT) protects the system from malicious attacks. Derived from the Byzantine Generals' Problem, which illustrates the challenge of achieving consensus among mutually distrustful parties, BFT mechanisms fortify distributed systems by enabling them to operate seamlessly even in the presence of faulty or malicious nodes. The Duality of Byzantine Fault Tolerance: A Double-Edged Sword

Byzantine Fault Tolerance (BFT) embodies a dual nature in the realm of distributed systems, presenting both positive and negative aspects to its implementation. This essay explores the dichotomy of BFT, highlighting its benefits and drawbacks in ensuring the reliability and integrity of distributed systems.

On the positive side, the implementation of BFT brings forth a robust defense mechanism against malicious attacks and system failures. BFT algorithms enable distributed nodes to coordinate and reach consensus, even in the presence of Byzantine faults. This capability ensures the integrity and consistency of the system, enhancing its resilience against adversarial conditions.

Moreover, BFT offers fault isolation and containment mechanisms, minimizing the impact of faulty or malicious nodes on the overall system. By detecting and isolating Byzantine faults, BFT prevents the spread of disruptive influences, thus maintaining the operational continuity of distributed systems. This aspect of BFT is crucial for ensuring system reliability and availability, especially in critical applications where uninterrupted operation is paramount.

However, the implementation of BFT also comes with inherent challenges and drawbacks. One notable negative aspect is the increased communication overhead associated with consensus protocols. BFT algorithms typically require multiple rounds of message exchanges among nodes to achieve agreement, resulting in elevated network traffic and latency. This overhead can impact the performance and scalability of distributed systems, particularly in large-scale environments or under high network load conditions.

Furthermore, achieving Byzantine Fault Tolerance often involves a trade-off between fault tolerance and system efficiency. While BFT mechanisms excel in ensuring fault tolerance and resilience, they may introduce complexities and resource overheads that can hinder system performance and scalability.

In our case we decided, that we do not any sort of BFT to have higher speed. The procedure we use to ensure strong consistency is already slowing down the system. In production and widely used system, implementing BFT is crucial concerning the security.

4.4 Voting

In order to select a leader as described in 2.1, a voting strategy is crucial. We chose the bully Algorithm to vote for a leader. In this algorithm, any node can start the election, but the node with the highest ID, whatever ID is chosen (e.g. Universally Unique Identifier (UUID) or IP Address), will win the election by "bullying" the nodes with a lower ID. If the comparison with any other ID is greater instead of greater or equal, if two nodes have the same ID, the node which voted first will win.

The Bully Algorithm provides a decentralized approach to electing a leader in a distributed system. By allowing nodes to autonomously determine the leader without relying on a central authority, the Bully Algorithm enhances system resilience and scalability. This decentralized nature ensures that the system remains operational even if certain nodes fail or become unreachable, thereby improving fault tolerance and system robustness.

However, one limitation of the Bully Algorithm is its susceptibility to network partitioning or communication failures. In scenarios where network partitions occur, nodes may incorrectly elect multiple leaders within different partitions, leading to inconsistencies and conflicts within the system. Additionally, if communication failures prevent nodes from exchanging messages during the leader election process, the algorithm may result in delays or failures to elect a leader, compromising the system's responsiveness and reliability.

We chose the Bully Algorithm for its decentralized approach to leader election, enhancing fault tolerance and scalability in distributed systems. As a bigger System is developped, an other algorithm should be implemented due to factors such as scalability limitations, susceptibility to network partitions, and the need for more complex fault tolerance mechanisms tailored to their specific requirements and workload characteristics.

4.5 Ordered Reliable Multicast

Johannes

5 Summary