Question Hub

Group 22: Jakob Waibel, Mouad Abuaisha, and Philipp Urban

1 Introduction

A group of independent computation devices connected by a network defines a distributed system. In order for components to interact with each other and schedule their actions so that users perceive the system as a single, integrated computing facility, each host carries out computations and runs a distribution middleware[4]. Distributed systems represent a significant advancement in computer science and IT. This is because an increasing number of related tasks are getting scaled up and more complicated and are beyond the capabilities of a single computer. Distributed systems reduce the risks associated with a single point of failure and therefore improve fault tolerance and reliability. Contemporary distributed systems are typically built with near-real-time scalability in mind. This allows for easy expansion and redistribution of computational resources, which boosts efficiency and shortens completion times. The architecture of a distributed system focuses on the system as a whole and how its components are distributed across several machines[5]. The selected distributed system's concept is explained in this proposal.

Question Hub is a distributed system for facilitating live Q&A sessions. Clients can ask and upvote questions anonymously in order to determine a ranking on the site and hence allowing them to ask questions they would be too afraid to ask otherwise. A lecturer can then answer questions according to that ranking during a lecture. Each client can vote for a question exactly once. Voting again removes the vote from the corresponding question.

2 Project Requirements Analysis

Architecture Model The project should implement a client-server[6] equipped with the ability to handle an arbitrary number of clients and servers. The servers should represent managers that handle the data flow and application state while ensuring the correct operation of the system. Each manager maintains a separate copy of the application data. Multiple servers exist in order to provide consistency and availability throughout the operation of the application. Each user is utilizing a client instance in order to post or vote for questions. Every action taken by a user will result in an update of the application state that is then propagated back to the clients.

Voting Voting shall be implemented in the form of a leader election algorithm. The leader election algorithm should regularly determine a server node that gets control of the application state. If the leader fails, a new leader should be determined immediately.

Dynamic Discovery of Hosts There can be an infinite number of clients and servers in the system. All nodes should be able to discover the currently leading node via broadcast. The control node should then notify the new node about the current network topology.

Fault tolerance The system should still be functional, if single server instances fail. If a leader fails, a new leader should be elected and take over the work from the previously elected leader.

Furthermore, the server nodes in the should regularly send heartbeat messages in order to determine liveness of the server nodes and detect possible node failures.

Ordered Reliable Multicast In order for the system to work in a consistent manner, it has to be guaranteed that messages originating from the same host are delivered in a similar order in which they were sent. In order to achieve this, first-in first-out reliable ordered multicast should be implemented in order to guarantee that the heartbeat messages can be processed currectly.

2.1 Dynamic Discovery

Host discovery works dynamically using broadcast messages. Whenever a new node joins the network, it sends a broadcast message to notify all other nodes about its presence.

In order to distinguish server and client nodes, the servers utilize the $HELLO$ OpCode, while the clients utilize the $HELLO\_SERVER$ OpCode. This is necessary since the different type of nodes are handled differently i.e. a client node does not have to be taken in account for possible future elections. By design, only the leader replies to $HELLO$ and $HELLO\_SERVER$ messages utilizing unicast by sending a $HELLO\_REPLY$ message containing the current topology of the network as well as the application state. This is possible since every nodes provides its unicast port as a field inside of every broadcast message. This provides the benefit of automatically announcing the current leader to the client as well as reducing the amount of messages in the system.

2.2 Fault Tolerance

In order to achieve fault tolerance, the system needs to distinguish multiple events that could lead to a failure. Liveness of the system is ensured by server nodes regularly broadcasting heartbeat messages to each other utilizing a push-based approach. These heartbeat messages ensure that every server node has a consistent topology of the system and hence can initiate a leader election once a leader crashes. A server is considered failed once a heartbeat that is normally sent every second was not received for a total of two seconds for a particular node. Once this event occurs, the node is removed from the topology and, if the node was the leader, the remaining nodes initiate a new leader election. If the

failed node was not a leader, the system removes the node from the topology and continues operation as normal.

If the server node reconnects at some point, it has to adhere to the communication protol again by broadcasting a message of OpCode $HELLO$. Whenever a new node joins, it challenges the current leader by initiating a new leader election.

Since the liveness of the system does not depend on client nodes, clients do not have to broadcast regular heartbeat messages. Since the questions are anoymous and are not less relevant when a client leaves, the application state does not have to be adjusted after a client crashed.

By identifying clients based on their socket, a client cannot vote for questions multiple times by reconnecting arbitrarily often, as long as it is using the same socket. A reconnecting client has to retransmit the message with OpCode $HELLO\_SERVER$ just like the server.

In an attempt to handle byzantine failures, every operation a client wants to perform is verified by the leading server instance before an operation is executed. After the client sends a unicast message containing the action to the leader and it has been verified, the server instance broadcasts the operation to all nodes. This leads to clients receiving the most up to date information in their respective frontend as well as replicating the application state to all other server nodes in case the client crashes.

A byzantine failure of a server instance can be tolerated by the system as long as the number of failed nodes does not exceed a third of the number of server instances in the system. This is implemented by utilizing the so-called "Oral Message" method message which is a proposed solution to handle such failures. In this system the leader sends the action to be performed to each other server instance. The other server instances then also act as a leader and send the message themselves to the other server instances. After executing this operation, each server instance received a message by every server instance and can decide based on the majority of messages received which action the system should perform. By using this approach, byzantine server instances can be tolerated as long as the constraint holds. [1].

2.3 Leader Election

Electing a leader is performed using the Hirschberg-Sinclair algorithm which resembles an algorithm for originally determinining the largest of n processors that are arranged in a ring. By utilizing the bidirectional capabilities of the Hirschberg-Sinclair algorithm, a performance of $O(n \log n)$ is possible.[2]

This algorithm was chosen since the number of participating nodes does not In General, the algorithm sends a message around in a ring of all nodes. Instead of comparing the size of processors, Question Hub is utilizing random UUIDs that are generated upon node creation. If a node receives a message from a node with a larger UUID, it just forwards that message onwards to the next node in the ring. If a sender receives its own messages of OpCode $ELECTION_VOTE$ again i.e. it had the largest UUID of all nodes, it won the election which will result

in the winning node broadcasting a message of OpCode $ELECTION_REPLY$. In order to achieve the optimum number of messages for the algorithm, the algorithm performs the election in rounds. The first round sends a message with hop-distance 1 to the left and to the right in the ring. In general, the hop distance can be described as sending messages a distance of 2^i hops in phase i. When the final hop of a given phase i is reached, the receiver sends a message with OpCode $ELECTION_REPLY$ back in the direction of the sender in order to notify the sender that the message received its destination without being dropped due to a node having a higher UUID than the sender. Hence, the sender can start another round of the election.

Whenever a new server node joins and receives a $HELLO\_SERVER$, it initiates a new leader election in order to challenge the current maximum UUID in the system in order to find out whether it should be the new leader.

When a node does not receive any messages of OpCode $HEARBEAT$ for 2 seconds and it is the only node in its node topology, it declares itself to be the leader.

Once a new leader was elected successfully, all nodes get a broadcast of OpCode $ELECTION_RESULT$ In order to notify the nodes about the new leader. Upon receiving this message, the clients switch their communication to the new leader.

2.4 Ordered Reliable Multicast

Since the application logic is always processed through the leader and the nodes get the application state from the leader at all times, ordered reliable multicast is not required in this scenario.

Instead, the heartbeat messages are using first-in first-out multicast in order to ensure that the servers receive the timestamps in right order. If messages arrived in a different order, timestamps could vary so much, that a node could appear as if it failed since the timestamp differs by more than two seconds.

FIFO is implemented by inserting arriving heartbeat messages into a queue and always delivering the message with the oldest heartbeat so that the heartbeat messages arrive in the correct order.

3 Architecture Design

Question Hub is implemented using a client-server [6] architecture model with multiple servers and clients. The servers represent managers that handle the data flow and application state while ensuring the correct operation of the system. Each manager maintains a separate copy of the application data. Multiple servers exist in order to provide consistency and availability throughout the operation of the application. Each user is utilizing a client instance in order to post or vote for questions. Every action taken by a user will result in an update of the application state.

The system can be scaled horizontally i.e. an arbitrary amount of n server nodes and m client nodes can participate. A client consists out of a frontend written using JavaScript and Vue as well as a middleware which exposes an HTTP API for the frontend to work with. The middleware itself aquires its data using socket communication with the currently leading server node. The server nodes interface with the clients using sockets and messages containing several different OpCodes e.g. $HELLO$, $VOTE\_REQUEST$ or $ELECTION\_RESULT$. In order to still deploy the client as one component, the subcomponents are containerized while still using the network of the host. By following this approach, the system can still be operated in a distributed manner.

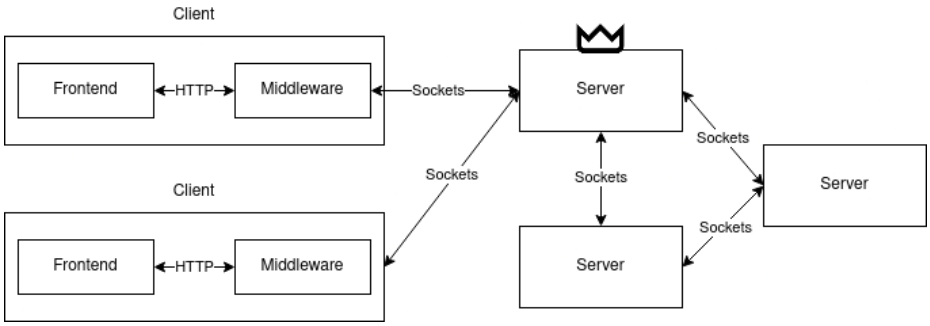

Fig. 1. System Design Architecture Diagram

4 Implementation

The system is comprised out of multiple components: The frontend, the middleware and the server. The frontend is implemented using JavaScript and Vue. The frontend fetches its data from a REST API that is exposed by the middleware. It provides a minimal set of endpoints that suffice to perform the business logic of the system:

- **GET**: */api/get* fetches the application data
- **POST**: */api/vote* votes for a question by providing the question id
- **POST**: */api/question* post a new question

Since the whole application data is fetched from the middleware whenever an update is aquired and the middleware is running on the same node as the frontend, additional API calls are not necessary.

The clients communicate with the servers using sockets. The server and the client middleware are implemented using Python. The REST API is implemented using Python's flask library. Multiple threads are utilized to listen on multiple channels simultaneously. One thread listens for broad- and multicast messages,

one thread listens for unicast messages and one is exclusively handling heartbeat messages.

The API between the client and the server adheres to the following OpCodes:

- $HELLO\_SERVER$ announces the clients presence to the server
- $HELLO\_REPLY$ acknowledges the announcement and contains the current application state for the client
- $QUESTION\_REQUEST$ sends a question request to the server
- $QUESTION$ acknowledges the question and broadcasts it to all nodes
- $VOTE\_REQUEST$ sends a vote request containing the corresponding question id to the server
- $VOTE$ acknowledges the vote and broadcasts it to all nodes
- $ELECTION\_RESULT$ broadcasts the election result to all nodes

The communication between servers requires some additional message types in order to address the leader election and fault tolerance:

- $HELLO$ to announce the servers presence
- $HEARTBEAT$ to send a liveness probe to the other server nodes
- $ELECTION\_VOTE$ to send a vote for a leader election
- $ELECTION\_REPLY$ to send a reply back to the sender of the vote
- $APPLICATION\_STATE$ to send the application state to a new server node

Architecturewise, the code is seperated into multiple components representing the several layers on the client and server-side. Each node contains a `ControlPlane` that contains the management of the topology of the network as well as the representation of the nodes as a ring for the leader election. Networking is handled using `Message`s that are responsible for composing, marshalling, unmarshalling and sending messages of several OpCodes.

Messages are transmitted utilizing JavaScript Object Notation (`JSON`). This allows for conveniently marshalling and unmarshalling messages and access fields across multiple languages.

Leader elections also compose a separate component, namely `Election` that is responsible for handling the election process.

5 Discussion and Conclusion

Question Hub forms a distributed system implementing several basic concepts e.g. leader election, dynamic discovery of hosts, fault tolerance. However, if the system would be utilized in a production environment, there are several points that could be improved. The current implementation utilizes a random UUID in order to determine a leader. Since the Hirschberg-Sinclair algorithms aims at determining a leader based on some metric like processor size or remaining disk-capacity. In a production environment, this metric could be useful and actually affect the performance of a large-scale distributed system.

Furthermore, additional ways of handling node failures might be useful for a production-grade system. Currently, only node crashes and invalid application state transitions are handled properly.

However, Question Hub provides a solid implementation for the scale of this project, implementing all of the set requirements.

References

1. Lamport, L., Shostak, R. & Pease, M. The Byzantine generals problem. *Concurrency: The Works Of Leslie Lamport*. pp. 203-226 (2019)
2. Hirschberg, D. & Sinclair, J. Decentralized extrema-finding in circular configurations of processors. *Communications Of The ACM*. **23**, 627-628 (1980)
3. Dommel, H.-P. and Garcia-Luna-Aceves, J.J. Ordered end-to-end multicast for distributed multimedia systems *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences* pp. 10 (2000)
4. Coulouris, G., Dollimore, J. & Kindberg, T. Distributed systems: concepts and design. (pearson education,2005)
5. Tanenbaum, A. Distributed systems principles and paradigms. (2007)
6. Berson, A. Client/Server Architecture 1996
7. Ongaro, D. & Ousterhout, J. In search of an understandable consensus algorithm. *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. pp. 305-319 (2014)
8. Lamport, L. The part-time parliament. *Concurrency: The Works Of Leslie Lamport*. pp. 277-317 (2019)