Question Hub

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

1 Introduction

Question Hub is a distributed system for facilitating live Q&A sessions. Clients can ask and upvote questions to determine a rank and therefore an order. Each client can vote for a question exactly once. Posting a question includes automatically upvoting it.

2 Architecture Model

Question Hub is implemented using a client-server architecture model. In the system an arbitrary amount of n server nodes and m client nodes can participate and it can be scaled horizontally. A client is resembled by the user interface as well as its backend consisting of the networking layer based on sockets required in order to communicate with the server instances. These instances communicate with other nodes using sockets. This communication includes handling events triggered by the client instances and managing the server instances. Each server instance contains a database that stores the current application state.

3 Dynamic Discovery

Host discovery works dynamically using broadcast messages. Whenever a new node joins the network, it sends a broadcast message with the $OpCode$ "HELLO" in order to notify all other nodes about its presence. Once a node receives the message, it updates its state, which is replicated inside of every node. In addition to the $OpCode$, the new node sends its IP and a port it listens on configured for unicast communication. The leading server node then answers the new node by sending a message with $OpCode$ "HELLO_REPLY" using unicast to the socket the new node has announced. This message contains the current state of the system at the time of sending the message out of the perspective of the leader.

4 Fault Tolerance

In order to achieve fault tolerance, the system needs to distinguish multiple events that could lead to a failure. All nodes are regularly broadcasting heartbeat messages in order to ensure liveness of a node and hence of the system. When a failure of a client instance is detected due to missing heartbeat messages for a given interval, the server instances remove the node from their list of clients so

that once a client reconnects, it needs to broadcast the initial "HELLO" message again in order to notify all host of its presence.

Since nodes are distinguished based on their IP addresses, a client changing its IP address and then reconnecting to the system will be handled as a new client. A reconnecting client keeping its IP address will be required to retransmit the "HELLO" message, but the votes the client made are preserved. If a client voted for a question in a previous session using the same IP address, it can't vote for the same question again.

A failure of a server instance can lead to different outcomes. If the server instance was the leader, a new leader needs to be elected. If the server instance was not the leader, the current leader can continue to operate the system. When the server node reconnects, it will get the delta of the data starting from the point where it crashed.

The application data is maintained by utilizing synchronized data replication between databases. In a leader election, a node can become the leader if and only if it has the most up to date data.

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 notifies others about the performed operation using a multicast message to all nodes.

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].

5 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 in a dezentralized, asynchronous way. By utilizing the bidirectional capabilities of the Hirschberg-Sinclair algorithm, a performance of $O(n \log n)$ is possible.[2] In General, the algorithm sends a message around in a ring of all nodes. If a node receives a message from a node with a larger processor, it just forwards that message onwards to the next node in the ring. Otherwise, it discards that message and sends its own message around to check whether it has the largest processor in the ring. If a sender receives its own message again i.e. it had the largest processor or all nodes, it won the election.

6 Ordered Reliable Multicast

In order to ensure that the number of votes of a certain question is represented correctly, the leader handles incoming vote requests using the First In, First Out method for achieving ordered reliable multicast when sending the messages to the rest of the system. This ensures that all nodes resemble the same state and no nodes are out of sync as long as no node failures occured.

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)