[What is consensus?]

Given a group of participants, how do you get them to all come to an agreement? This could be the outcome of some voting, or maybe a decision reached by a single elected nominee.

This is the fundamental problem faced by participants when reaching for a consensus.

No matter the method, decision-making involving consensus ensures that all members collaborate and cooperate to reach a single outcome without any conflict.

[Why consensus algorithms are necessary and how do they apply to distributed systems?]

This proposition can be extended to, and is a necessary factor in distributed computer systems, where multiple independent systems have to be able to reliably communicate and come to single agreement.

Consensus algorithms allow for the coordination of multiple processes across different machines by enforcing consistency between individual decisions. Furthermore, integrity must be maintained, with reliability against any faults that can affect individual process uptime and have security against potential tampering.

[Applications of consensus algorithms]

Currently consensus algorithms are commonly used in scenarios with high replication, such as blockchain networks to correctly log transactions, and with systems that require replicated state machines, such as load balancers, because they require multiple nodes to coordinate and form a single contiguous system.

[Consensus algorithm properties]

There are three properties that a consensus algorithm must satisfy:

The first is that every process must terminate and decide on a single value. They cannot decide on multiple values or run indefinitely.

Secondly, if all processes propose the same value, integrity is maintained if they all decide on that value. There can be no interference once a valid proposition has been made.

And finally, to reach consensus, it is essential that each process agrees on the same, single value.

[Challenges consensus algorithms have to address]

Consensus algorithms have to be fault-tolerant, and be resistant to Byzantine faults, by maintaining data integrity, and crashes, by making sure the abrupt stopping of a single node does not break the entire system.

Challenges such as asynchronous communication can affect the reliability of a system. Additionally, it makes it harder to determine whether a response will arrive from a process, possibly violating the first property of consensus algorithms, where a process always has to terminate.

Network permissions can also pose a challenge for consensus algorithms to mitigate, as if users can join the system on-demand, it becomes vulnerable to malicious inflation of processes, threatening to impact the decision-making process and affect the result.

[What is Raft?]

To implement consensus, we are going to explore and study Raft for this project. It is named after the properties the algorithm tries to address, and is designed to be reliable, replicated, redundant and fault-tolerant.

The algorithm describes a method to replicate state transitions within a system, to ensure each node receives a consistent set of instructions in their logs, and to retain system uptime after any unexpected faults in individual nodes that cause connection loss.

[Background Survey]

Raft is a method to introduce a replicated state machine over a cluster of nodes, allowing each machine to consistently and safely replicate their logs.

An alternative is Paxos, an algorithm that has the same goals in data consistency and fault tolerance, but prioritises safety over Raft's liveness with a more complicated agreement process. Furthermore, Raft was found to be more lightweight than Paxos with a dedicated leader to handle log replication.

Other alternatives include the Viewstamped Replication or other atomic broadcast algorithms.

[Log replication]

In Raft, we have an elected leader.

This leader makes its presence known to its followers using a heartbeat.

When a follower wants to make a transaction, it sends the request to the leader..

Then the leader relays the transaction to every other follower. Once a majority of followers confirm that this change has been received, the leader commits this change to their log and tells each follower to also commit this change. This ensures that each of the logs on every node remains consistent in an asynchronous environment.

[Leader election]

In the case a leader crashes,

a new leader must be chosen to handle log replication.

Firstly, followers with the most up-to-date logs become candidates. After this is determined,

each node will be able to cast a single vote for a candidate to determine the new leader.

The candidate with the majority of votes wins the election to become the new leader of the system.

This ensures fault-tolerance against any crashes or any errors that affect the responsiveness of the leader nodes, allowing the system to retain some protection against independent failures.

[21 - Multi-person Chat Room]
Using the knowledge of Raft Consensus, we introduce our proposed application.
A Multi-person chat room app,

[22 - Features]
The app has three features:

1. Group chat room: Members can exchange real time messages within their respective groups
2. Leaders role: When a user creates a group, they become the leader of that group and have the authority to accept or reject requests from other users who want to join.
3. Multiple Groups: The app can have multiple chat groups, where each user only belongs to one group

[23 - Illustration]
Here is the illustration of the app structure. We have nodes that synchronize in Raft consensus. These nodes act as servers. For the sake of this presentation only, we will use 5 nodes

[24 - Illustration]
The app user will connect and act as a client. They will be served by a worker node

[25 - Illustration]
Another user comes in and is also served by a worker node. In this example somehow they are served by the same worker node

[26 - Illustration]
New client connects to the server and might be served by different node this time.

[27 - Illustration]
The nodes work as a load balancer. So the node used to serve the client can be different from one another, regardless of the client groups. Meaning clients in the same group can be served by different nodes and vice versa

[28 - Raft support the App]
Raft consensus is a good fit for implementing the app server.

1. Raft can provide logical clocks through heartbeats. This can be used as a message timestamp, ensuring message order.
2. This structure ensures fault tolerance by allowing another server to take over in case of failure.
3. Enforcing a limitation on group creation to prevent overload, by limiting the number of groups with respect to the number of nodes.

[Notes]

- Raft is *not* Byzantine fault tolerant as the nodes trust an elected leader