Preliminary Notes

This class recalls background on distributed systems and cryptography, essential building blocks for studying blockchain technology. This laboratory is based on several sources [1–7]. We recommend students to conduct additional research on other sources [8, 9].

1 Distributed Systems

A Distributed System is a system comprised of software and hardware components that are connected on a network, coordinating their actions via message exchange.

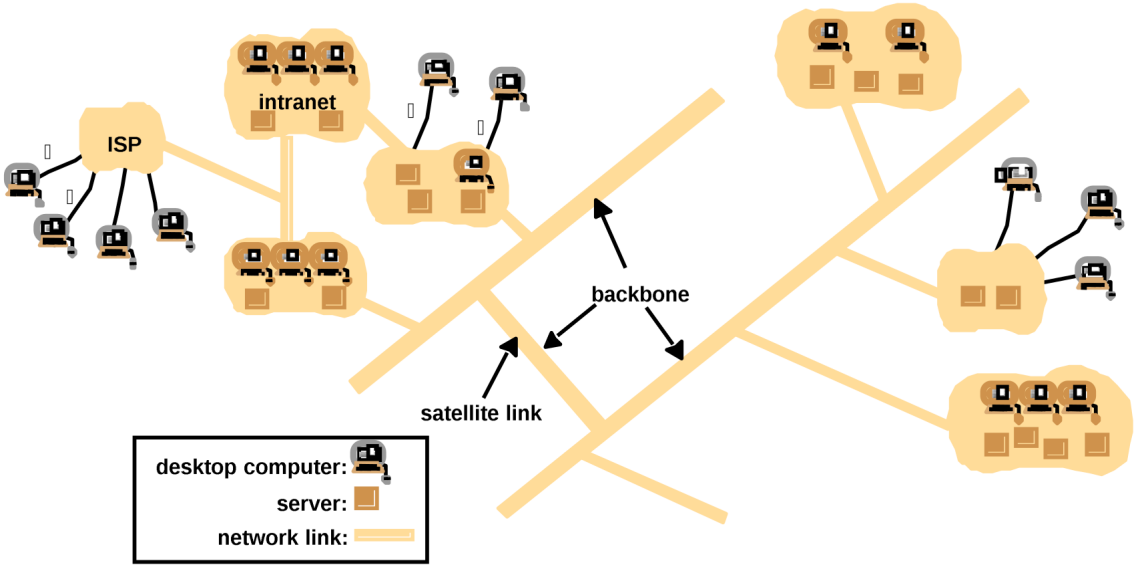


Figure 1: Enterprise IT system [3]

Nowadays, most modern enterprise information systems are distributed systems, popularized due to the Internet's growth.

Servers communicate with other servers using protocols, such as the HTTP protocol, based on the TCP/IP protocol. Communication can also happen locally (i.e., between computer processes, using interfaces such as sockets). Typically, such protocols are independent of the system architecture.

In fact, there are different architectures for organizing distributed systems. One of the simplest is the client-server architecture, where servers can expose endpoints to clients. In this architecture, servers keep resources and serve those resources according to requests made by clients. Client-server architectures have a popularized programming model: RPC, and the more recent gRPC. Such protocols allow structuring distributed system programming based on client calls on the server's code. While this is an efficient

scheme, it constitutes a centralized paradigm that suffers from a lack of scalability and constitutes a single point of failure. Other architectures offer different tradeoffs: for instance, the peer-to-peer architecture is decentralized, which allows for a higher degree of openness and fault tolerance. This makes the whole system more robust to faults.

1.1 Faults, Errors, and Failures

A fault is an event that alters the expected behavior of a system, which typically causes errors. Errors comprise the transition from a correct state of the system to an incorrect state. Errors can provoke failures if the system deviates from its specification. The protection against faults allows participants to remain consistent: this way, it is easier to maintain a correct state shared with other participants. Faults can be defined as crash faults or byzantine faults. Crash faults are faults where a node becomes unresponsive. This can be due to either physical problems, like a power outage or a faulty component, or networking problem, like network partitions, leading to the node being unreachable. Byzantine faults occur when a node acts arbitrarily and strays from the specification. This can include fake information, different responses to different nodes, or even a completely random answer.

1.2 Fault Tolerant Models

There are different ways to deal with each type of fault. Crash Fault Tolerant (CFT) models can usually withstand up to $\frac{N}{2}$ crashes, with N being the total number of nodes. This model assumes a quorum of $\frac{n}{2} + 1$ nodes, which has to agree on the global state. This means that as long as there is a **majority of nodes operational**, a consistent answer can be returned.

However, suppose one of the nodes is malicious. In that case, the system may enter a faulty (or inconsistent) state with the node giving fake information. As such, for Byzantine Fault Tolerant (BFT) models, we need at least $3f + 1$ correct nodes to support f malicious nodes. This way, we can guarantee that we have at least a **majority of correct nodes** answers in any given quorum.

But a question may arise: how to make sure that the participants agree on a single state? How can we assure agreement under uncertain conditions and assuming that participants can fail (e.g., connection problems)? What if there are malicious participants? This problem is also referred to as the consensus problem, a well-known problem studied in the early stages of distributed systems.

1.3 Consensus

Consensus is a fundamental problem in distributed systems, in the area of fault tolerance. It involves multiple servers (or nodes) agreeing on a global state (set of values).

Pease, Shostak, and Lamport's Byzantine generals story popularized the problem of reaching consensus, where the generals had to agree on a common plan to attack a city. In this scenario, messengers that exchanged their messages were unreliable [10], because they could be captured or change sides.

Distributed systems ought to be *Byzantine fault-tolerant*, as there may be malicious nodes on the network [6]. A consensus algorithm is used to create agreement on a global ledger state in the presence of crash faults or Byzantine faults. Consensus algorithms depend on the assumptions made about the environment and the system that the algorithm is running. For instance, what are the faults that can happen (crash vs byzantine), how are messages passed, synchronous (communication and processing delays are bounded) vs asynchronous (communication and processing delays are not bounded, i.e., may take an arbitrary amount of time to be completed) environment.

1.4 State Machine Replication

A general approach to building fault-tolerant systems is state machine replication. A state machine can be described by a set of possible initial states, accepting states, and a transition function. This function determines the possible steps, given the current state.

In a distributed system, each participating node has a state machine and a log. A log is a collection of log entries, which typically contain timestamped information. The state machine (for example, a computer program) stores and manipulates state (for example, the state can be values are written in a hash table or an array with commands to be executed).

The idea of state machine replication is that clients can interact with a distributed system that agrees upon a global state while it appears that it is interacting with a single state machine. Even if one of the servers fails, the system still responds to the requests. Each node takes commands from the logs, triggering the execution of a function. The consensus ensures that the commands received are the same while ensuring a common ordering for the commands. This ensures that each state machine yields the same series of results, achieving the same series of states, in a common order. Each state machine executes the same function calls, in the same order, yielding the same output. For this reason, state machines need to be deterministic (otherwise, the same input could produce different results, causing inconsistencies in the system).

Figure 2 shows an example of a replicated state machine. A group of machines executes a deterministic function based on a replicated log they receive (input). The idea is that logs are kept identical, so the execution on different machines is performed using the same commands and in the same order. When the first machine receives an update request (set variable Z to 6), it first replicates that command into all the logs, passing that command to the other machines on the cluster. All machines add the newest command to the end of their logs. After that, the command can be executed by each state machine. This triggers the original state machine to return the answer

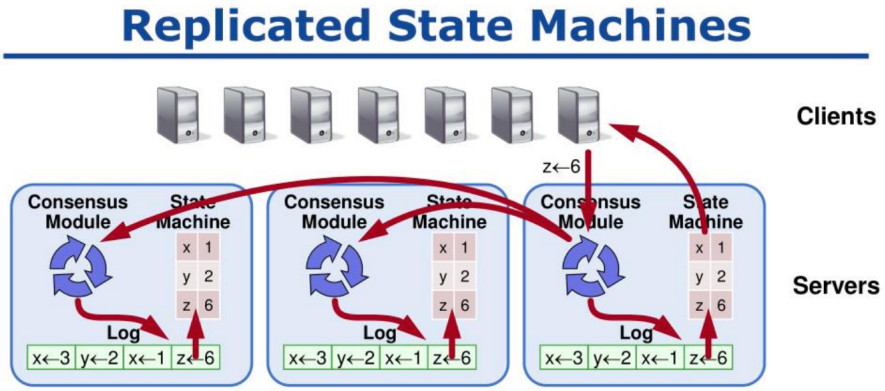


Figure 2: Replicated state machines [11]

to the execution to the client. If one of the servers (that comprises a state machine) fails, the other servers can still process the request, as far as a majority of servers are operational. There are consensus algorithms that apply either Crash Fault Tolerant or Byzantine Fault Tolerant models. These systems allow us to develop and deliver resilient, dependable systems that can satisfy today's needs in terms of availability and scalability.

With this in mind, consensus algorithms aim to replicate commands into the replicated logs, allowing us to assure desirable properties about a system, like validity, agreement, and termination.

A system is valid if the value decided by the system was a value proposed by a node. This way, we avoid cases where a node can inject a decision that no one has seen. Moreover, a system agrees if no two correct processes decide differently (e.g., output different values for the same input). Termination is a liveness property that states "every correct process eventually decide". Those are all desirable properties for fault-tolerant systems.

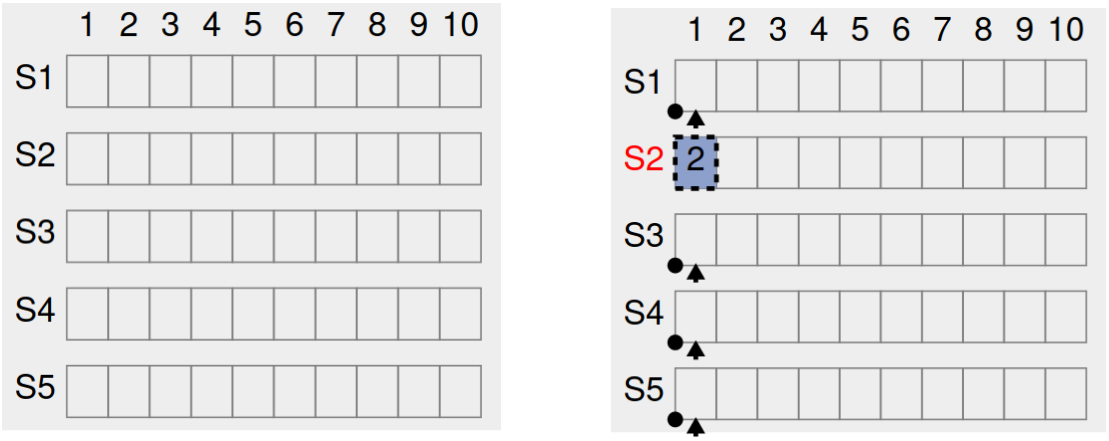
In this class, you will work in greater detail the RAFT algorithm: a well-adopted consensus algorithm, which focuses on understandability and has many practical applications, inclusively on the blockchain area.

1.5 The RAFT consensus algorithm

RAFT is a crash fault-tolerant consensus algorithm that is designed to be easy to understand. It belongs to a class of consensus algorithms called leader-based (or coordinator-based or primary-backup). In leader-based consensus, a leader calculates a decision on the state and communicates it to the followers; if the leader is faulty, a new leader is elected.

RAFT considers several participants:

- leader: the node that receives client requests, manages the log replication and



(a) Initial state of the shared log of operations

(b) Leader S2 received a command from a client. It wishes to updated state

Figure 3: States at different terms for each server

sends log update requests to other nodes. While having a leader that commands the global state is sufficient fine in some scenarios, a malicious leader can constitute a problem;

- Followers: constituting all nodes except the leader;
- Candidates: followers requesting to be the leader.

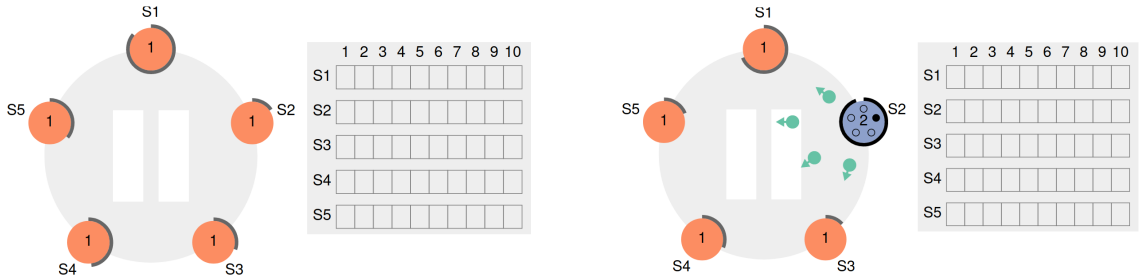
As explained before, a distributed system built with RAFT has several servers running a RAFT instance and a persistent log to store the commands received from clients. The replication of logs and consequent command execution is given by three phases:

1. Leader Election
2. Log Replication
3. Safety

Figure 3 shows the global state, which corresponds to the state of each node (server) across the different terms (time periods in which the algorithm executes). Terms are used to detect obsolete information: there is at most one leader per term, and followers do not accept information coming from a previous term than the one they record. This way, the algorithm guarantees that each node has the latest information.

1.5.1 Leader Election

Time in RAFT is divided into terms. Each term has a unique identifier, a monotonically increasing number set to 0 at server startup time. RAFT uses randomized timers to elect



(a) There is no leader, each node awaits instructions

(b) After the timeout, the node sends a proposal to become the new leader

Figure 4: Leader election process

the leader, illustrated in Figure 4. To know if the leader is alive, RAFT uses a heartbeat system. At the beginning of the consensus process, nodes are awaiting heartbeats from the leader (Figure 4a). When a follower receives a heartbeat, their timeouts are reset. As there is no leader, no heartbeats are received within the election timeout, and eventually, a timeout is triggered. The node that initiated the timeout sends a proposal to become the new leader to every other node (Figure 4b), and eventually becomes the new leader.

Suppose a node does not receive a heartbeat from a leader within a certain time frame (for example, in case the leader crashes). In that case, there is another timeout that initiates the beginning of a new term, and a new leader election process starts.

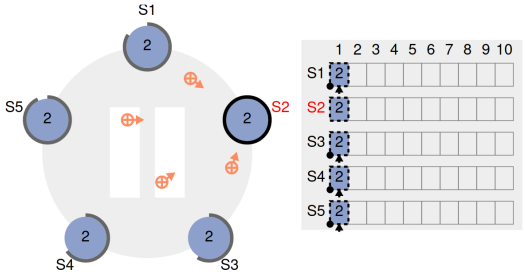
The terms also allow for liveness: nodes reject requests coming with an invalid term number (older than the current term). Nodes ask to be a leader on a first-come-first-served basis. As the majority rule applies, only one node can win an election and become the leader. In case of a draw, another round of election proceeds. Note that the log is empty at the beginning of the execution (Figure 3).

1.5.2 Log Replication

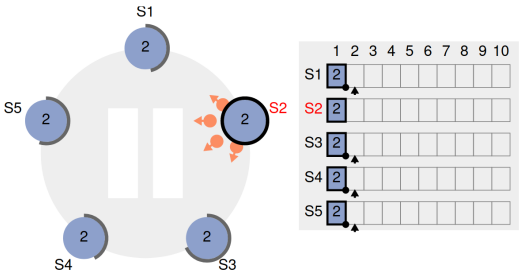
The idea of log replication is the leader to accept requests from clients, which translates to commands that the server has to execute. Those commands are appended to a log. After that, the leader replicates its logs to other nodes on the network to execute the received commands and keep the states of all nodes consistent. Part of this procedure is illustrated in Figure 5.

Let us now suppose that a leader receives a request from a client (Figure 5a). The log is updated, but it is still not safe to execute the received commands (as the log entry is considered uncommitted). This happens because followers are not aware of the request (illustrated by the dashed lines around an update). The leader broadcasts the most recent log entry to their followers and receives a message from each one acknowledging that they received the updated log entry (Figure 5b). After the leader

confirms the state, after receiving the followers' acknowledgments, the followers can append the command to their logs, for later executing the command.



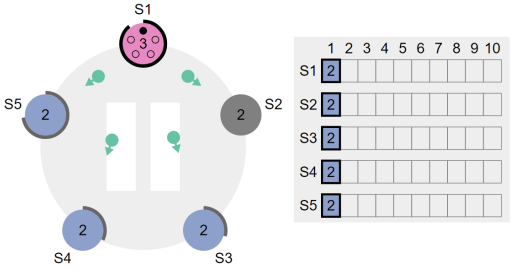
(a) Followers acknowledge the latest log entry



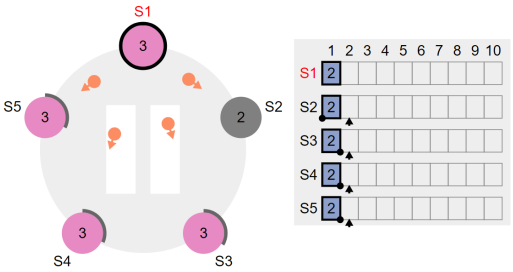
(b) Global status is committed, that is, the received command from the client is present in the logs of the majority of nodes, so each node can now execute it

Figure 5: State updates

Imagine that now, node S2 crashes due to a hardware failure. This causes that S2 cannot send heartbeats back to each follower. We now assume that S1 timeouts - it becomes a candidate and tries to become a leader (Figure 6a). When a follower becomes a candidate, it increments the current term, votes for itself, and send requests to other followers to become the new leader. Eventually, S1 wins the election and becomes the new leader (Figure 6b).



(a) Server S2 crashes. A follower triggers a timeout, becoming a candidate. The candidate S1 tries to become the leader



(b) Candidate S1 becomes the leader

Figure 6: New leader election process

1.5.3 Safety

Logs need to be kept consistent, and, furthermore, only nodes with the latest log version can become leaders. If a crash occurs, log inconsistencies may happen. In other words, the information stored by each node may not be the same at the same point. To solve

this problem, the log of the current leader is always considered the correct version. The leader repairs inconsistencies, if applicable. For that, there cannot be missing log entries, and log entries need to be ordered.

RAFT's approach to maintaining consistency is by matching logs and solving conflicts: each time a leader sends a log entry to a follower, it includes the term number and log index of the preceding entry. If there is a match, the follower accepts the new log entry; otherwise, the follower rejects the log entry. In case a log entry is rejected, the leader tries again, sending the current log entry, the preceding one, and the one further before., until a match happens. When a match happens, the follower's log is updated, according to the leader.

The explained mechanisms guarantee the log matching property: all previous entries are committed if a given entry is committed. If two logs share the same log index and term, then those logs are identical (represent the same log entry). Lastly, the desirable safety tackles leader completeness: once a log entry is committed, the leader will have that log entry stored. This assures a committed entry is never lost. Nodes with incomplete logs are not chosen in the election process. Essentially, a node denies the vote if their log is more complete than the candidate's log.

2 Hands on RAFT

Now that we illustrated the RAFT algorithm, let's consolidate that knowledge. It is recommended for students to read the additional support materials available: [7, 12]. You may use the simulator provided at RAFT's homepage to aid you in your responses[1].

Exercise 1:

To consolidate your knowledge, watch the guided visualization of the RAFT algorithm [13]. Elaborate on how the RAFT algorithm can increase the system's availability from a client perspective.

Exercise 2:

When does a new term start?

Exercise 3:

How do candidates act if two of them attempt to become leaders at precisely the same time?

[1] https://RAFT.github.io/

Exercise 4:

If a follower receives a message that has a different term than the one recorded, does it accept the message?

Exercise 5:

Suppose that a leader receives a command from a client and notifies all followers. Upon sending the second transaction, one of the followers crashes. How does RAFT guarantee that the follower is updated?

Exercise 6:

Consider the following RAFT cluster, composed by a leader and four followers. The leader has sent eight messages, corresponding to eight commands. Those eight commands translate into eight log entries. The current term is term three.

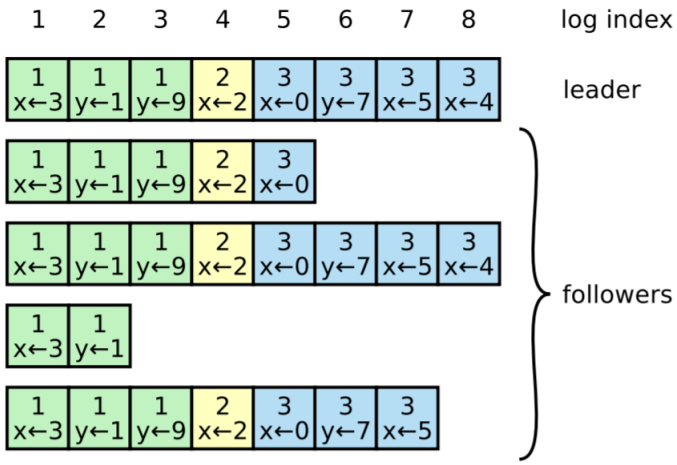


Figure 7: RAFT cluster composed of 5 machines

What are the commands, represented by each log index, that are safe to be executed by each node?

Exercise 7:

Consider the following RAFT cluster, composed of a leader and four followers. There are inconsistencies in the logs across nodes. Assume that node S4 crashed on term 1. Node S5 became the leader on term 2, but was only successful in replicating logs 4 and 5, before crashing. Node S3 is the leader in term 3.

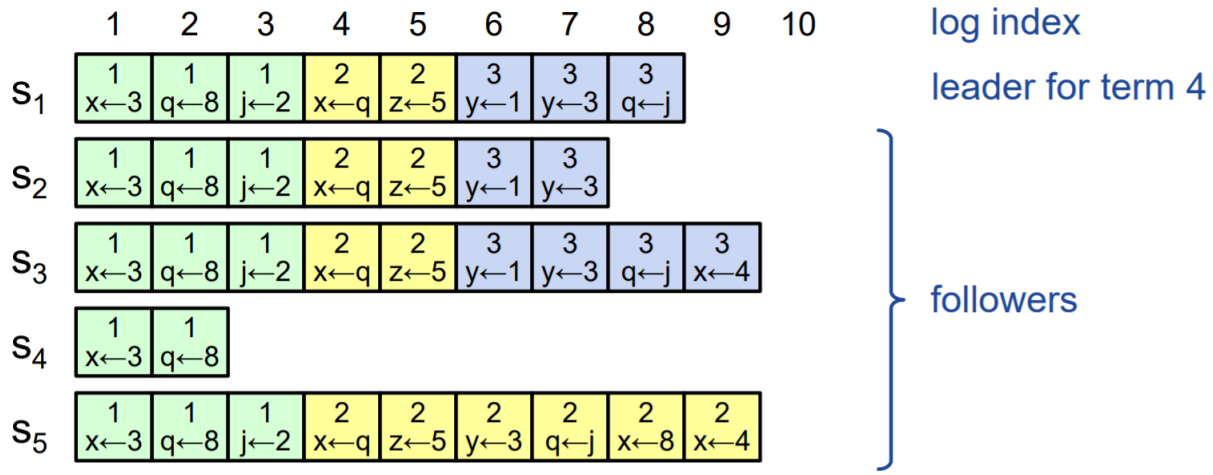


Figure 8: RAFT cluster composed of 5 servers, showing inconsistencies regarding the log

What will be the system's final state, assuming no new commands are received, and that S5 recovers from a crash?

Exercise 8:

During the leader election process, a mechanism that ensures consistency across logs is the rejection of candidates with outdated logs. What are the checks the follower does against the candidate's proposal? On which occasion does it reject voting on the candidate?

Exercise 9:

Consider the following RAFT cluster, composed of seven nodes.

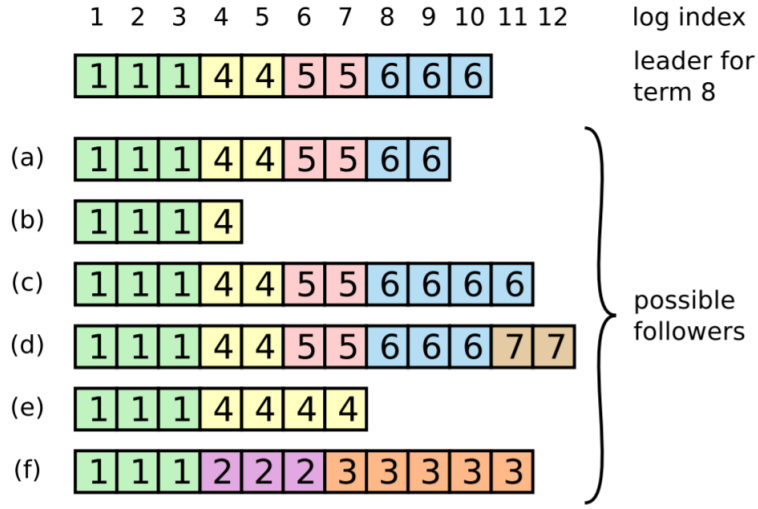


Figure 9: RAFT cluster composed of 7 nodes

Given that the leader of term eight has ten log entries and interacts with six followers, how did the system get into this state?

Exercise 10:

How does the system re-organize itself if a node leaves the system?

References

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. USA: Addison-Wesley Publishing Company, 2011.

[2] Técnico Lisboa, "Initial Page · Distributed Systems," 2017. [Online]. Available: https://fenix.tecnico.ulisboa.pt/disciplinas/SDis12645111326/2016-2017/2-semestre

[3] B. Wong, "CS454/654 Distributed Systems," 2014. [Online]. Available: https://cs.uwaterloo.ca/~bernard/courses/cs454/0.Begin.pdf

[4] J. Ousterhout and D. Ongaro, "Designing for Understandability: The Raft Consensus Algorithm - YouTube," 2016. [Online]. Available: https://www.youtube.com/watch?v=vYp4LYbnnW8

[5] P. Veriissimo and L. Rodrigues, *Distributed systems for system architects*, 2001. [Online]. Available: http://www.google.com/books?hl=en{&}lr={&}id=oOzwLX1{_}bpkC{&}oi=fnd{&}pg=PR13{&}dq=Distributed+systems+for+system+architects{&}ots=AoTDFJClxL{&}sig=AZVUrUivDNOc{_}hFwZLB9S2zrKYQ

[6] M. Correia, "From Byzantine Consensus to Blockchain Consensus," *Essentials of Blockchain Technology*, p. 41, 2019.

[7] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX Annual Technical Conference*, 2014.

[8] "6.824: Distributed systems - mit." [Online]. Available: https://pdos.csail.mit.edu/6.824/

[9] "Cos-418, fall 2016: Distributed systems - princeton." [Online]. Available: https://www.cs.princeton.edu/courses/archive/fall16/cos418/index.html

[10] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," Tech. Rep., 1982. [Online]. Available: https://people.eecs.berkeley.edu/{~}luca/cs174/byzantine.pdf

[11] D. Ongaro and J. Ousterhout, "Replicated state machines." [Online]. Available: https://image3.slideserve.com/5547558/replicated-state-machines-l.jpg

[12] RAFT, "Raft Consensus Algorithm," 2016. [Online]. Available: https://raft.github.io/

[13] B. Johnson, "Raft - Understandable Distributed Consensus," 2013. [Online]. Available: http://thesecretlivesofdata.com/raft/