CMPSCI 677 Operating Systems **Spring 2019**

Lecture 19: April 3

Lecturer: Prashant Shenoy *Scribe:* ***Justin Svegliato (2019), Disha (2018)***

Announcements

- The midterm grading has started. It should be done in a few days.
- Homework 3 has been released. It involves using different functions of Amazon Web Services. Be sure to request credentials for your student account on Amazon Web Services as soon as possible.
- On Monday, there will be a guest lecture on pervasive computing, IoT, and smart buildings.
- There will no class on Wednesday.

19.1 Byzantine Fault Tolerance

In the last lecture, we introduced fault tolerance. Among a handful of topics, we discussed a range of different types of faults. The two main types of *node failures* that were discussed are described below.

A **crash fault** is when a process crashes but works properly up until it crashes. In order to tolerate f crash faults, we need $f + 1$ processes. For example, consider a system with 2 processes that work properly. Now suppose that one of those processes crashes for some reason. Naturally, we only need 2 processes since we know that the other process of the system works properly by assumption. This is the easiest fault to handle.

A **Byzantine fault** is when a process produces arbitrary responses at arbitrary times. This could even be the result of a malicious adversary. In order to reach agreement when there are f Byzantine faults, we need $3f + 1$ processes. Intuitively, we need a two-thirds majority to reach agreement and eliminate the faulty process (as seen in the **Byzantine Generals Problem** discussed last lecture). As a result, Byzantine fault tolerance is substantially more expensive because we need 3 times the amount of processes. Solely detecting Byzantine faults, however, only requires $2f + 1$ processes. This is the hardest fault to handle.

19.2 Reaching Agreement

In a distributed system, **reaching agreement** means that every node agrees on the result of a given computation. It is important to note that *reaching agreement* differs from *tolerating faults*. In the case of tolerating faults, the system needs the ability to *tolerate* nodes that crash. However, in the case of reaching agreement, assuming that every node is up, the system needs the ability to *agree* on the result of a computation. This means that reaching agreement requires more than one node by definition.

As a side note, when the delivery of a message is unbounded in terms of time, it is not possible for agreement to be reached because a slow process would be indistinguishable from a faulty process.

BAR fault tolerance is a model where nodes can be Byzantine, altruistic (i.e., they can help faulty nodes like taking requests), and rational (i.e., they can perform operations like reporting timeouts).

19.3 Reliable Communication

Although we have only discussed node failures up until now, there can of course be *network failures*. We discuss how to address network failures in two types of communication below.

19.3.1 Reliable One-One Communication

In **one-one communication**, there is communication between a single sender (the client) and a single receiver (the server). This can be viewed as **unicast**. There is a range of network failures that come up in one-one communication:

1. The client may be unable to locate the server.
2. There may be lost request messages that do not reach the server.
3. The server may crash after receiving a request.
4. There may be lost reply messages that do not reach the client.
5. The client may crash after sending a request.

When there is a network failure in one-one communication, it can be handled in one of two ways:

- Use reliable transport protocols like TCP to handle network failures (2) and (4)
- Add additional logic to the application layer to handle network failures (1), (3), and (5)

Note that an earlier lecture on remote procedure call (RPC) semantics discussed most of this in more detail.

19.3.2 Reliable One-Many Communication

In **one-many communication**, there is communication between a single sender and multiple receivers. This can be viewed as **multicast**. When a message—or a packet—between the sender and one of the receivers has been lost, there is a network failure. In response, the sender must resend the message to the receiver. There are two basic schemes often used to handle network failures that come up in one-many communication:

- **ACK-based Schemes**: All receivers send an acknowledgement for every message received
- **NACK-based Schemes**: All receivers send a negative acknowledgement for any message not received

19.3.2.1 ACK-based Schemes

An **ACK-based scheme** is a generalization of TCP to one-many communication. In this approach, a sender first sends a message to every receiver in the multicast group (see Figure 19.1). When a receiver receives the message, it then sends an acknowledgement (an `ACK` message) back to the sender indicating that it received that message (or missed an earlier message in some implementations) (see Figure 19.2). The sender in turn waits to receive an acknowledgement from every receiver. Finally, if the sender does not receive an acknowledgement from a specific receiver after some timeout, it resends the message back to that receiver.

Drawback. Although such an approach is effective for one-one communication, it does not scale to one-many communication under certain conditions. For instance, if the multicast group were to have 10000 nodes, the sender would be flooded with 10000 acknowledgements after sending a message to each receiver of the multicast group. The sender would then need to process 10000 acknowledgements in order to keep track of and resend messages to specific receivers. The sender therefore becomes a *bottleneck* because it is infeasible to maintain such a large multicast group. This problem is typically called **ACK explosion**.

19.3.2.2 NACK-based Schemes

A **NACK-based scheme** is a more scalable approach to one-many communication that addresses the sender becoming a bottleneck due to ACK explosion. In this approach, a sender once again sends a message to every receiver in the multicast group (see Figure 19.1). However, when a receiver receives a message, it does nothing (rather than sending an acknowledgement back to the sender like in the case of an ACK-based scheme). Instead, the receiver sends a negative acknowledgement (a `NACK` message) back to the sender when it determines that it has missed an earlier message (see Figure 19.3). For example, if a receiver receives message i followed by message $i + 2$, it will send a negative acknowledgement back to the sender indicating that it missed message $i + 1$. A negative acknowledgement can also be sent to the neighbors of the receiver. If a neighbor has the missing message stored in its buffer, it can forward that message to the receiver. **Since every receiver is not flooding the sender with acknowledgements, NACK-based schemes are usually more scalable than ACK-based schemes.**

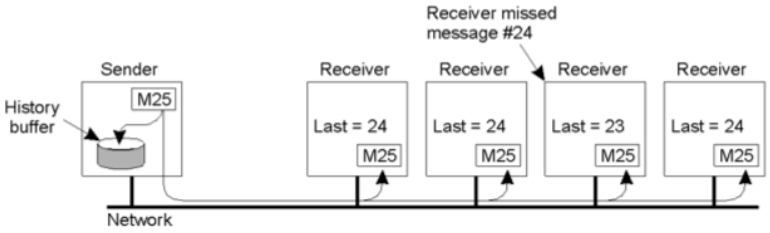


Figure 19.1: The sender sends a message to all receivers

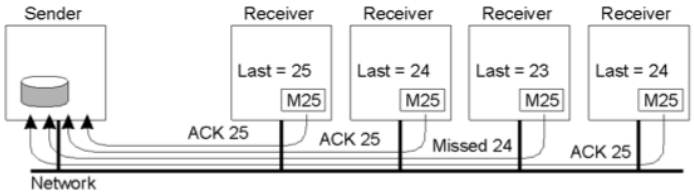


Figure 19.2: All receivers send an acknowledgement back to the sender

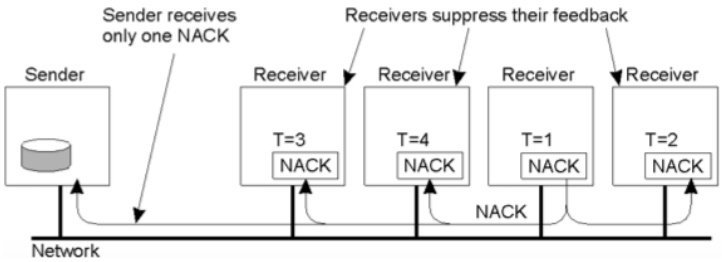


Figure 19.3: A receiver sends a negative acknowledgement back to the sender

Question. What if messages are out of order?

Answer. A NACK-based scheme will not work. We assume in-order message delivery. Otherwise, we have to use an ACK-based scheme.

Question. Instead of sending a bunch of `ACK`s, can a receiver send an `ACK` every nth message indicating that it received the last n messages?

Answer. Certainly. This has been tried already. If you have 10000 nodes, the sender will still get a lot of `ACK`s. If the network is doing its job, a NACK-based scheme will still be more efficient (though the sender will get a lot of `NACK`s if the network is lossy). There are advantages and disadvantages to both schemes based on the properties of the network.

Question. What if the receiver receives a message that it reported as missing at a much later point?

Answer. If we assume in-order message delivery, this will never happen.

Question. Does this only work across LANs? Or does this also work across the Internet?

Answer. It depends. There is a protocol called *IP multicast* that takes TCP/IP, which is unicast, and extends it to a multicast setting that can be used over WANs. This has not been deployed anywhere because it makes the router stateful. Therefore, while it can be done, there is no real deployment of such a protocol.

Question. Are there protocols other than TCP/IP that provide in-order message delivery?

Answer. While there are many stacks that have been developed, they are not used in place of TCP/IP.

19.4 Atomic Multicast

Atomic multicast guarantees that a message is received by either *all* of the processes or *none* of the processes of a multicast group. In other words, when one process sends a message to a group of processes, either every process or none of the processes receive that message. It should not be possible for some processes to have the message and some processes to not have the message.

Problem. How do we handle when a process in the multicast group crashes?

Solution. We will use a **group view** where every message is uniquely associated with a group of processes. Each process will have the same view of the group and only send to processes within that group. The basic idea of Figure 19.4 on how to use a group view to handle a process crashing is below:

- Suppose a process sends a message to every process within a group.
- The system must make sure that either *all* or *none* of the processes in the group receive the message.
- When a message is sent to every process in the group, there are two possible outcomes:
  - If all processes receive that message, the system continues to run without any problems.
  - However, if one process does not receive the message as a result of crashing, the system has to stop the message from being delivered to every other process in the group. Most importantly, the process that crashed should then be removed from the group.

Example. Atomic multicast can be useful in replicated databases. Suppose we have 3 replicas of a database. When an update is sent to each replica, we want to guarantee that all or none of the replicas of the database

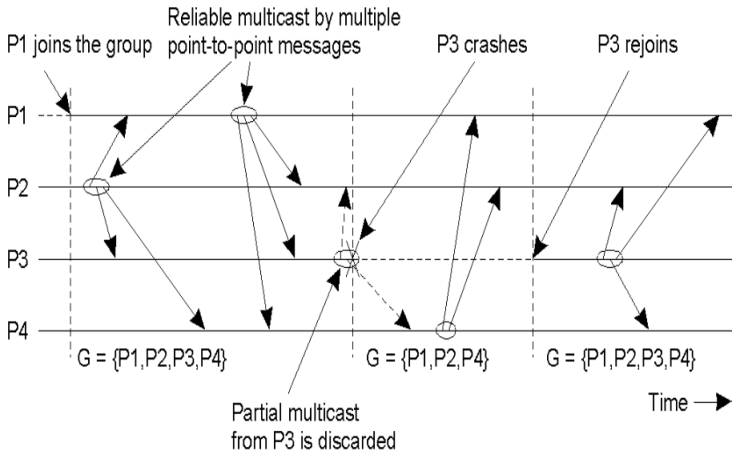


Figure 19.4: An example of atomic multicast for a group of processes

receive the update. If one of the replicas were to not receive the update unlike the other replicas, the database would be left in an inconsistent state. Atomicity is therefore critical to replicated databases.

Note that the slides *Implementing Virtual Synchrony in Isis* (Slide 17) and *Implementing Virtual Synchrony* (Slide 18) were skipped during the lecture.

19.5 Distributed Commit

Atomic multicast is an example of a more general problem. In a **distributed commit**, every process in a group either performs or does not perform a given operation. An operation can range from appending to a log file, delivering a message to a multicast group, or updating a database. We discuss two variations of a distributed commit involving a different number of phases below.

19.5.1 Two Phase Commit

In a **two phase commit**, there is a *coordinator* that coordinates an operation within a group of *subordinates*. To reiterate, for a given operation, either all or none of the subordinates perform that operation. Note that the node that is a coordinator is also a subordinate. The phases of a two phase commit are as follows:

- **Voting Phase:** All subordinates vote on whether to commit or abort the operation
- **Decision Phase:** All subordinates commit or abort the operation given the results of the voting phase

In general, a two phase commit can be broken down into a series of steps:

1. The coordinator asks every subordinate to vote on whether to commit or abort a given operation.
2. Each subordinate sends its vote back to the coordinator:
   (a) If a subordinate sends a commit vote, it waits for the results of voting from the coordinator.
   (b) However, if a subordinate sends an abort vote, it just aborts that operation. This is because the subordinate knows that all other subordinates must abort the operation as well.

3. The coordinator gradually collects each vote from the subordinates.
4. The coordinator reports the results of voting back to the subordinates:
   (a) If there are only commit votes, the coordinator tells the subordinates to commit.
   (b) However, if there is at least one abort vote, the coordinator tells the subordinates to abort.
5. All subordinates either commit or abort the operation and send an acknowledgement to the coordinator.

19.5.1.1 Coordinator Perspective

We summarize the role of the *coordinator* during a *two phase commit* in Figure 19.5a below:

- The coordinator waits for a `Commit` message in the `INIT` state.
- The coordinator sends a `Vote-request` message to the subordinates.
- In the `WAIT` state, the coordinator waits for the subordinates to respond to the `Vote-request` message.
- Depending on the responses from the subordinates, the coordinator can do one of two things:
  - If the coordinator receives at least one `Vote-abort` message, it issues a `Global-abort` message to the subordinates and then moves to the `ABORT` state.
  - However, if the coordinator receives only `Vote-commit` messages, it issues a `Global-commit` message to the subordinates and then moves to the `COMMIT` state.

19.5.1.2 Subordinate Perspective

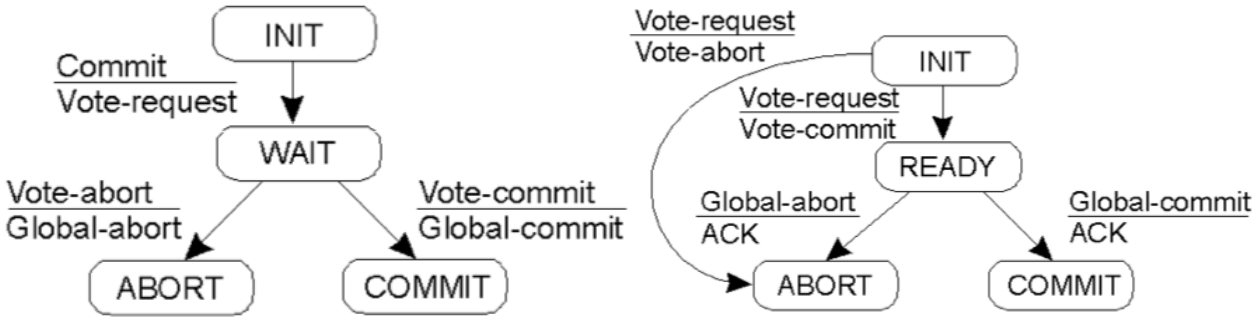
We summarize the role of a *subordinate* during a *two phase commit* in Figure 19.5b below:

- In the `INIT` state, a subordinate waits for a `Vote-request` message from the coordinator.
- Upon receiving a `Vote-request` message from the coordinator, the subordinate either sends a `Vote-commit` message or a `Vote-abort` message back to the coordinator depending on if it can perform the operation:
  - If the subordinate replies with a `Vote-commit` message, it moves to the `READY` state.
  - However, if the subordinate replies with a `Vote-abort` message, it moves to the `ABORT` state. This is because the subordinate knows that all other subordinates must abort as well.
- Once the subordinate is in the `READY` state, it waits for a `Global-abort` message or a `Global-commit` message that moves it to the `ABORT` state or the `COMMIT` state respectively.

19.5.1.3 Crash Failure Recovery

We discuss two kinds of crash failures that can occur during a two phase commit below.

Subordinate Crash Failure. *How do we handle when a subordinate crashes?* Suppose that a subordinate crashes during a two phase commit. When the subordinate recovers, it will be in one of the following states:



(a) The role of the coordinator

(b) The role of a subordinate

Figure 19.5: The flow of a two phase commit

- `INIT`: If the subordinate recovers in the `INIT` state, it has not voted yet. As a result, the coordinator will still be waiting for a vote from the subordinate. Therefore, the coordinator should tell all other subordinates to abort. When the subordinate finally comes back up, it will see that all other subordinates have aborted and then abort as well. Another way to do this is to just have the subordinate send an abort vote when it finally comes back up.
- `ABORT`: If the subordinate recovers in the `ABORT` state, it has aborted the operation already. The coordinator would have then already issued an abort to all other subordinates given the abort vote of this subordinate. Therefore, the subordinate can stay in this state or move to the `INIT` state.
- `COMMIT`: If the subordinate recovers in the `COMMIT` state, it has committed the operation already. The coordinate would have then already issued a commit to all other subordinates given the commit vote of this subordinate. Therefore, the subordinate can stay in this state or move to the `INIT` state.
- `READY`: If the subordinate recovers in the `READY` state, it has sent a commit vote to the coordinator already. However, once the coordinator receives every other vote, it will either issue an abort or a commit to all other subordinates depending on the results of the vote. When the subordinate finally recovers, it will not know what to do. It must therefore ask a neighbor for the results of the vote.

Coordinator Crash Failure. *How do we handle when the coordinator crashes?* This is where a two phase commit has some trouble. If the coordinator fails during the voting phase, every subordinate can either get stuck in deadlock or be inconsistent. For instance, suppose that the coordinator calls for a vote. The subordinates in turn vote to commit the operation and then move to the `READY` state. Unfortunately, however, the coordinator receives every vote but crashes before sending the results of the vote to the subordinates.

Can the subordinates determine what to do? Nah, here are some options that all run into problems:

- **The subordinates can ask each other if they voted to commit or abort the transaction.** This is unsafe because the node that is a coordinator is also a subordinate. Suppose that all subordinates agree to commit the transaction. If the coordinator had decided to abort the transaction, the system will transition to an inconsistent state where every subordinate commits the transaction even though the coordinator had aborted the transaction.
- **The subordinates can abort the transaction.** This is unsafe because the node that is a coordinator is yet again also a subordinate. Suppose that all subordinates originally voted to commit the transaction. The coordinator then committed the transaction but crashed before it could send the outcome of voting to all other subordinates. In this scenario, the coordinator would have committed the transaction while the subordinates would have aborted the transaction.

- **Wait for the coordinator to come back up.** If the coordinator never comes back, the subordinates will be deadlocked. While this preserves the safety property, it does not satisfy the liveness property.

Question. How does this differ from a distributed snapshot?

Answer. A distributed snapshot is a way for all of the nodes to save their state in a consistent way. A distributed commit is a generalization that can be extended to other operations like appending to a log file or updating a database.

Question. Does the approach used by a distributed snapshot work for a distributed commit?

Answer. The big difference between a distributed snapshot and a distributed commit is that we did not assume that there could be node failures in a distributed snapshot. We want a distributed commit to still work in the presence of node failures.

Question. Under what conditions can a subordinate go from the `INIT` state to the `ABORT` state directly?

Answer. This will only happen when the subordinate votes to abort the operation.

Question. Why would a transaction be aborted?

Answer. A transaction can be aborted for any reason. Maybe something went wrong with it. The reason why a transaction has been aborted should not be a concern here.

Question. Can a vote have an expiration time?

Answer. It could. The problems related to the coordinator do not go away though.

Question. Is the coordinator a single point of failure?

Answer. Yes, we will talk about this with Paxos.

Note that the slides *Implementing Two-Phase Commit* (Slide 21) and *Implementing 2PC* (Slide 22) were skipped during the lecture. Moreover, the slide *Recovering from a Crash* was skipped because it was discussed prior to reaching that slide.

19.5.2 Three Phase Commit

In order to handle coordinator crash failures, a **three phase commit** adds a **precommit phase** to the phases of a two phase commit. This involves adding a `PRECOMMIT` state between the `WAIT` state and the `COMMIT` state in the coordinator and between the `READY` state and the `COMMIT` state in a subordinate. We see that everything stays the same aside from a few highlighted steps that have been added or modified below.

19.5.2.1 Coordinator Perspective

We summarize the role of a *coordinator* during a *three phase commit* in Figure 19.6b below:

- The coordinator waits for a `Commit` message in the `INIT` state.
- The coordinator sends a `Vote-request` message to the subordinates.
- In the `WAIT` state, the coordinator waits for the subordinates to respond to the `Vote-request` message.
- Depending on the responses from the subordinates, the coordinator can do one of two things:

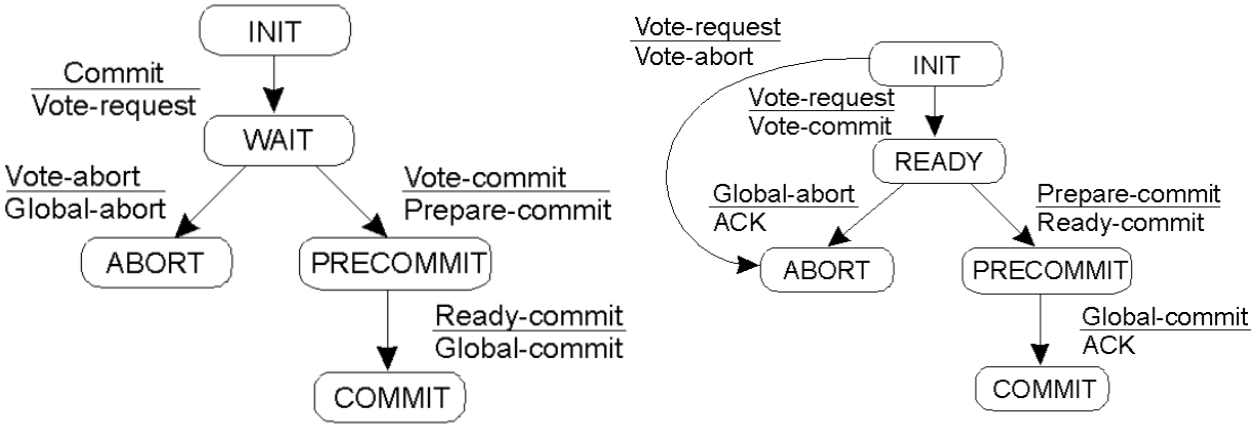
  - If the coordinator receives at least one `Vote-abort` message, it issues a `Global-abort` message to the subordinates and then moves to the `ABORT` state.
  - However, if the coordinator receives only `Vote-commit` messages, it issues a `Prepare-commit` message to the subordinates and then moves to the `PRECOMMIT` state.

- Once the coordinator receives a `Ready-commit` message from the subordinates, it issues a `Global-commit` message to the subordinates and then moves to the `COMMIT` state.

19.5.2.2 Subordinate Perspective

We summarize the role of a *subordinate* during a *three phase commit* in Figure 19.6b below:

- In the `INIT` state, a subordinate waits for a `Vote-request` message from the coordinator.
- Upon receiving a `Vote-request` message from the coordinator, the subordinate either sends a `Vote-commit` message or a `Vote-abort` message back to the coordinator depending on if it can perform the operation:
  - If the subordinate replies with a `Vote-commit` message, it moves to the `READY` state.
  - However, if the subordinate replies with a `Vote-abort` message, it moves to the `ABORT` state. This is because the subordinate knows that all other subordinates must abort as well.
- Once the subordinate is in the `READY` state, it waits for a `Global-abort` message or a `Prepare-commit` message that moves it to the `ABORT` state or the `PRECOMMIT` state respectively.
- The subordinate waits in the `PRECOMMIT` state until receiving a `Global-commit` message from the coordinator that moves it to the `COMMIT` state.



(a) The role of the coordinator

(b) The role of a subordinate

Figure 19.6: The flow of a three phase commit

19.5.2.3 Crash Failure Recovery

How does this handle coordinator crash failures? Suppose that the coordinator asks for a vote and then receives a vote from every subordinate. This means that all subordinates have moved to the `READY` state. Now let us assume that the coordinator crashes and so the subordinates never receive the results of

voting from the coordinator. In this scenario, it would be safe for the subordinates to abort the operation. If the coordinator were to come back up, it would be in the `PRECOMMIT` state rather than the `COMMIT` state. This means that the coordinator would not have committed the transaction yet. Once the coordinator finally comes back up, the subordinates can tell the coordinator that the transaction has been aborted.

Question. Should there be a transition from the `PRECOMMIT` state to the `ABORT` state?

Answer. Yes. The diagram does not show what happens in the case of failure though.

19.6 Replication for Fault Tolerance

As an aside, replication can be used for two different purposes. It is possible to use replicas of a service for either performance reasons (**Technique 1**) or fault tolerance reasons (**Technique 2**). Note that this lecture has mostly focused on replication for fault tolerance.

Technique 1: A request can be split between different replicas of a service

- If one replica fails, we can send requests to other replicas.
- This is suitable for crash fault tolerance because we can use another replica when one replica crashes.

Technique 2: A request can be sent to all replicas of a service

- All replicas vote on their results and take the majority result.
- This is suitable for Byzantine fault tolerance.
- Examples of this technique include 2PC, 3PC, and Paxos.

19.7 Consensus and Agreement

Agreement vs. Byzantine Agreement vs. Consensus

- **Agreement**: All processes must agree on the result of some computation or to perform some operation (like a commit, an abort, a leader election, or a database update). If there is a problem with a process, this will not work because there cannot be total agreement.
- **Byzantine Agreement**: A coordinator makes a decision based on the majority vote of every process (i.e., the Byzantine Generals Problem). If there is a problem with the coordinator, this will not work.
- **Consensus**: All processes act as a coordinator and ask all other processes for a majority vote. This is a group decision and does not require a coordinator. Note that there can only be crash faults here.

Properties of a Consensus Protocol

- **Agreement**: Every nonfaulty process must *agree* on the same value.
- **Termination**: Every nonfaulty process must *decide* on some value.
- **Validity**: If all processes *propose* a value, all nonfaulty processes must *decide* on that value.

- **Integrity**:
  - Every nonfaulty process *decides* at most one value.
  - If a nonfaulty process *decides* on a value, a process must have *proposed* that value.

19.8 2PC/3PC Problems

Both two phase commits and three phase commits experience problems in the presence of different types of failures. While the **safety** property can be ensured, the **liveness** properties cannot always be guaranteed due to node failures and network failurs: the system will never perform an operation that leads to an inconsistent state (satisfying the safety property) but can still be deadlocked (violating the liveness property). We describe a few caveats associated with each type of commit below.

Two Phase Commit

- It must wait for the coordinator and the subordinates to be running.
- It requires all nodes to vote.
- It requires the coordinator to always be running.

Three Phase Commit

- It can handle coordinator failures.
- But network failures are still a problem.

There has been an implicit assumption that there could only be node failures instead of network failures during a two or three phase commit. While a node could crash in the network, the network would never experience any issues. Suppose, however, that the network was partitioned into two partitions due to some problem. Although both partitions will continue to function correctly, each partition cannot communicate with each other. **By definition, if the network is partitioned due to some problem, a two or three phase commit cannot work because every node is required to vote on the answer.**

In order to eliminate such an assumption, we have to revisit the definition of *agreement*. Rather than requiring the vote of every node, we can just require the vote of the majority of nodes. Therefore, if the network were to be separated into two partitions, the partition with the majority of nodes can still continue to function properly. This idea forms the basis of **Paxos**, a consensus protocol. **Instead of requiring every node to vote, Paxos only requires the majority of nodes to vote.**

19.9 Paxos Requirements

Paxos satisfies the following properties:

- Safety (*Correctness*)
  - All nodes must agree on the same value.
  - The agreed upon value must be computed by some node.

- Liveness (*Fault Tolerance*)
  - If less than $\frac{n}{2}$ nodes fail, the remaining nodes will eventually reach agreement. This allows the system to make progress in the presence of failures.
  - Note that that liveness is not guaranteed if there is a steady stream of failures as the protocol determines what to do. If a node fails in the middle of the protocol, it must be restarted.
- **Why is agreement hard?** Because even in the face of failures, we still need to reach agreement.
  - The network might be partitioned.
  - The leader may crash during solicitation or before announcing the outcome of voting. While the current round will not produce any results, a new leader will be elected through leader election. All nodes will then vote again.
  - A new leader may propose different values from the value that had been agreed upon originally.
  - Several nodes may become a leader at the same time. This is possible when the network is partitioned due to a network failure. The left half will elect a new leader while the right half will have the old leader, and they will still continue to function properly. Both sides of the partition may agree on different things unfortunately.

19.10 Paxos Introduction

Paxos is a 3 phase protocol. Similar to the other approaches that have been discussed, we have a leader. The leader proposes a solution to some computation to every node in the system. Rather than asking the nodes whether or not the transaction should be committed or aborted, we instead ask the nodes whether they agree or disagree with the proposed solution. If the majority of nodes agree with the answer, the leader will determine that the answer is correct. However, if the majority of the nodes disagree with the answer, we repeat this process. Each node can also propose its own solution (though this was not discussed in detail). As a side note, the leader only has to wait for the majority of nodes to vote instead of all of the nodes.

Note that the majority of the slides on Paxos were skipped and saved for the next lecture.