DISTRIBUTED SYSTEMS

Contents

Part 1. Consistent global states

1. Asynchronous distributed systems

A **distributed system** is a collection of sequential processes $p_1, p_2, \ldots, p_n$ and a network capable of implementing unidirectional communication **channels** between pairs of processes for message exchange. Channels are reliable but may deliver messages out of order. We assume that every process can communicate with every other process, perhaps through intermediary processes.

An **asynchronous system** is characterized by the following properties: there exist no bounds on the relative speeds of processes and there exist no bounds on message delays. Asynchronous systems rule out the possibility of processes maintaining synchronized local clocks or reasoning based on global real-time. Communication remains the only possible mechanism for synchronization in such systems.

2. Distributed computations

A **distributed computation** describes the execution of a distributed program by a collection of processes. The activity of each sequential process is modeled as executing a sequence of **events**. An event may be either internal to a process and cause only a local state change, or it may involve communication with another process. In an asynchronous distributed system where no global time frame exists, events of a computation can be ordered only based on the notion of "happen-before": if e sends message m and e' receives message m, then $e \to e'$. In general, the only conclusion that can be drawn from $e \to e'$ is that the mere occurrence of e' and its outcome may have been influenced by event e. It is possible that for some e and e', neither $e \to e'$ nor $e' \to e$: we call such events concurrent.

2.1. **Global states, cuts and runs.** Let σ_i^k denote the **local state** of process p_i immediately after having executed event e_i^k and let σ_i^0 be its initial state before any events are executed. The **global state** of a distributed computation is a set of local states, one for each process. A **cut** is a set of local states, one for each process. A **run** of a distributed computation is a total ordering R where, for each process p_i, the events of p_i appear in the same order that they appear in its local history h_i. A single distributed computation may have many runs, each corresponding to a different execution.

In asynchronous distributed systems, the global state obtained through remote observations could be obsolete, incomplete, or inconsistent. It should be clear that uncertainties in message delays and in relative speeds at which local computations proceed prevent a process from drawing conclusions about the instantaneous global state of the system to which it belongs. Another source of difficulty in distributed systems arises when separate processes independently construct global states. The variability in message delays could lead to these separate processes constructing different global states for the same computation.

2.2. **Monitoring distributed computations.** For the time being, we will assume that a single process p_0 called the **monitor** is responsible for evaluating the global state of a distributed system. In the first strategy we pursue for constructing global states, the monitor p_0 takes on an active role and queries each process about its state. Upon the receipt of such a message, p_i replies with its current local state σ_i. When all n processes have replied, p_0 can construct the global state $(\sigma_1, ..., \sigma_n)$.

Note that the positions in the process local histories that state enquiry messages are received effectively defines a cut. Given that the monitor process is part of the distributed system and is subject to the same uncertainties as any other process, the above approach is not sound. While every cut of a distributed computation corresponds to a global state, only certain cuts correspond to global states that could have taken place during a run.

2.3. **Consistency.** A cut is **consistent** if for all events e and e'

$$(e' \in C) \wedge (e \rightarrow e') \Rightarrow e \in C$$

A consistent global state is one corresponding to a consistent cut. Similarly, notions such as "before" and "after" that are defined with respect to a given time in sequential systems have to be interpreted with respect to consistent cuts in distributed system: an event e is before (after) a cut C if e is to the left (right) of the frontier of C.

3. Observing distributed computations

Let us consider an alternative strategy for the monitor p_0 in constructing global states based on a reactive architecture. In this approach, p_0 will assume a passive role in that it will not send any messages of its own. The other processes, however, will be modified slightly so that whenever they execute an event, they notify p_0 by sending it a message describing the event. The monitor constructs an **observation** of the underlying distributed computation as the sequence of events corresponding to the order in which the notification messages arrive. We note certain properties of observations as constructed above

(1) due to the variability of the notification message delays, a single run of a distributed computation may have different observations at different monitors
(2) an observation can correspond to a consistent run, an inconsistent run or no run at all since events from the same process may be observed in an order different from their local history. A consistent observation is one that corresponds to a consistent run

There is the possibility of messages being reordered by channels that leads to undesirable observations. We can restore order to messages between pairs of processes by defining a delivery rule for deciding when received messages are to be presented to the application process. Communication from process p_i to p_j is said to satisfy **FIFO delivery** if, for all messages m and m',

$$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m')$$

While FIFO delivery is sufficient to guarantee that observations correspond to runs, it is not sufficient to guarantee consistent observations.

3.1. **Global real-time clocks.** Initially, assume that all processes have access to a **global real-time clock** and that all message delays are bounded by δ. Let $RC(e)$ denote the value of the global clock when event e is executed. When a process notifies p_0 of some local event e, it includes $RC(e)$ in the notification message as a timestamp. Let **DR1** be the delivery rule employed by p_0: At time t, deliver all received messages with timestamps up to $t - \delta$ in increasing timestamp order.

To see why an observation O constructed by p_0 using DR1 is guaranteed to be consistent, first note that an event e is observed before event e' if and only if $RC(e) < RC(e')$. This is true because messages are delivered in increasing timestamp order and delivering only messages with timestamps up to time $t - \delta$ ensures that no future message can arrive with a timestamp smaller than any of the messages already delivered. Since the observation O coincides with the delivery order, it is consistent iff the **clock condition**

$$e \to e' \Rightarrow RC(e) < RC(e')$$

is respected (satisfied because timestamps are generated using the global real-time clock).

3.2. **Logical clocks.** In an asynchronous system where no global real-time clock can exist, we can devise a simple clock mechanism for "timing" such that event orderings based on increasing clock values are guaranteed to be consistent with causal precedence. In other words, the clock condition can be satisfied in an asynchronous system.

Each process maintains a local variable LC called its **logical clock** that maps events to the positive natural numbers. The value of the logical clock when event e_i is executed by process p_i is denoted $LC(e_i)$. Each message m that is sent contains a timestamp $TS(m)$ which is the logical clock value associated with the sending event. Before any events are executed, all processes initialize their logical clocks to zero. The following update rules define how the logical clock is modified by p_i with the occurrence of each new event

$$LC(e_i) = \begin{cases} LC + 1 & \text{if } e_i \text{ is an internal or send event} \\ max\{LC, TS(m)\} + 1 & \text{if } e_i \text{ is a receive event} \end{cases}$$

In other words, when a receive event is executed, the logical clock is updated to be greater than both the previous local value and the timestamp of the incoming message. Otherwise the logical clock is simply incremented. It is easy to verify that for any two events where $e \to e'$ the logical clocks associated with them are such that $LC(e) < LC(e')$. Thus, logical clocks satisfy the clock condition of the previous section.

Since each logical clock is monotone increasing and FIFO delivery preserves order among messages sent by a single process, when p_0 receives a message m from process p_i with timestamp $TS(m)$, it is certain that no other message m' can arrive from p_i such that $TS(m') \leq TS(m)$. This leads to a new delivery rule **DR2** for constructing consistent observations (when using logical clocks): deliver all received messages that are stable at p_0 in increasing timestamp order.

3.3. **Causal delivery.** Recall that FIFO delivery guarantees order to be preserved among messages sent by the same process. A more general abstraction extends this ordering to all messages that are causally related, even if they are sent by different processes. The resulting property is called **causal delivery**: for all messages $m,\ m'$, sending processes $p_i\ p_j$, and destination process p_k

$$send_i(m) \to send_j(m') \Rightarrow deliver_k(m) \to deliver_k(m')$$

In other words, in a system respecting causal delivery, a process cannot known about the existence of a message any earlier than the event corresponding to the delivery of that message. Note that having FIFO delivery between all pairs of processes

is not sufficient to guarantee causal delivery. The relevance of causal delivery to the construction of consistent observations is obvious: if p_0 uses a delivery rule satisfying CD, then all of its observations will be consistent. The correctness of this result is an immediate consequence of the definition of CD, which coincides with that of a consistent observation.

3.3.1. *Constructing the causal precedence relation.* For implementing causal delivery efficiently, what is really needed is an effective procedure for deciding the following: given events e, e', that are causally related and their clock values, does there exist some other event e'' such that $e \to e'' \to e'$ (i.e. e'' falls in the causal "gap" between e and e')? Given $RC(e) < RC(e')$ (or $LC(e) < LC(e')$), it may be that e causally precedes e' or that they are concurrent. What is known for certain is that $\neg(e' \to e)$. Having just received the notification of event e', DR1 and DR2 could unnecessarily delay its delivery even if they could predict the timestamps of all notifications yet to be received. The delay would be unnecessary if there existed future notifications with smaller timestamps. We strengthen the clock condition by adding an implication in the other sense to obtain **strong clock condition**:

$$e \to e' \Leftrightarrow TC(e) < TC(e')$$

where TC is a new timing mechanism.

3.3.2. *Causal histories.* A brute-force approach to satisfying the strong clock condition is to devise a timing mechanism that produces the set of all events that causally precede an event as its "clock" value. We define the causal history of event in distributed computation as the set

$$\theta(e) = \{e' | e' \to e\}$$

In other words, the causal history of event is the smallest consistent cut that includes e. When causal histories are used as clock values, the strong clock condition can be satisfied if we interpret clock comparison as set inclusion. From the definition of causal histories, it follows that

$$e \to e' \Leftrightarrow \theta(e) \subset \theta(e')$$

The unfortunate property of causal histories that renders them impractical is that they grow rapidly.

3.4. **Vector clocks.** Causal history can be represented as a fixed-dimensional vector rather than a set. Note that $\theta_i(e) = h_i^k$ for a process p_i and, since $\theta(e) = \theta_1(e) \cup ...\theta_n(e)$, the entire causal history can be represented by an n-dimensional vector $VC(e)$ where for all $1 \leq i \leq n$, the ith component is defined as

$$VC(e)[i] = k, \quad \text{iff } \theta_i(e) = h_i^k$$

Each process maintains a local **vector clock** VC of natural numbers where $VC(e_i)$ denotes the vector clock value of p_i when it executes event e_i. Each process initializes VC to contain all zeros. The following update rules define how the vector clock is modified by p_i with the occurrence of each new event e_i:

$$\begin{cases} VC(e_i)[i] = VC[i] + 1 & \text{if } e_i \text{ is an internal or send event} \\ VC(e_i) = max\{VC, TS(m)\} & \text{if } e_i = receive(m) \end{cases}$$

In other words, an internal or send event simply increments the local component of the vector clock. A receive event, on the other hand, first updates the vector

clock to be greater than (on a component-by-component basis) both the previous value and the timestamp of the incoming message, and then increments the local component. Given the above implementation, the jth component of the vector clock of process p_i has the following operational interpretation for all $j \neq i$:

$$VC(e_i)[j] \equiv \text{number of events of } p_j \text{ that causally precede event } e_i \text{ of } p_i$$

Definition. Property 1 (strong clock condition)

$$e \to e' \equiv VC(e) < VC(e')$$

3.4.1. *Implementing causal delivery.* Assume that processes send a notification message to the monitor for all of their events. As usual, each message m carries a timestamp $TS(m)$ wich is the vector clock value of the event being notified by m. A message m from process p_j is deliverable as soon as p_0 can verify that there are no other messages whose sending causally precede that of m. Let m' be the last message delivered from process p_k, where $k \neq j$. Before message m of process p_j can be delivered, p_0 must verify two conditions:

(1) there is no earlier message from p_j that is undelivered
(2) there is no undelivered message m'' from p_k such that

$$m' \to m'' \to m, \quad \forall k \neq j$$

No undelivered message m'' exists if $TS(m')[k] \geq TS(m)[k]$ for all k. This test can be efficiently implemented if p_0 maintains an array $D[1...n]$ of counters, initially set to zeros, such that $D[i] = TS(m_i)[i]$ where m_i is the last message that has been delivered from process p_i. Then we can formulate **DR3** (causal delivery) as: deliver message m from process p_j as soon as both of the following conditions are satisfied

$$\begin{cases} D[j] = TS(m)[j] - 1 \\ D[k] = TS(m)[k] \qquad \forall k \neq j \end{cases}$$

When p_0 delivers m, array D is updated by setting $D[j]$ to $TS(m)[j]$.

3.5. **Distributed snapshots.** We will now develop a strategy where p_0 requests the states of the other processes and then combines them into a consistent global state, assuming for semplicity that channels implement FIFO delivery.

3.5.1. *Snapshot protocol.* Initially, assume that all processes have acces to a real-time clock RC and that all message delays are bound by δ. Process p_0 chooses a time t far enough in the future in order to guarantee that a message sent now will be received by all other processes befor t

(1) At time t_0, process p_0 sends a snapshot-request message to all processes, including a time $t \geq t_0 + \delta$
(2) when real-time clock RC reads t, each process p_i records its local state σ_i

Being based on a real-time clock, it is easy to see that this protocol constructs a consistent global state

$$(e \to e') \wedge (e' \in C) \to e \in C$$

From this, we can develop the **Chandy-Lamport protocol**, which doesn't need real-time clocks nor logical clocks

(1) process p_0 sends a snapshot-request message to all processes

(2) let p_s be the process from which p_i receives the snapshot-request message for the first time. Upon receiving this message, p_i records its local state σ_i and relays the snapshot-request message to each other process
(3) let p_f be the process from which p_i receives the snapshot-request message for the last time. Since process p_i has received this message from each other processes, its participation in the snapshot protocol can now end

The Chandy-Lamport protocol requires $n \cdot (n+1) = O(n^2)$ messages.

3.5.2. *Properties of snapshots.* Let S_0 be the global state in which the snapshot protocol is initiated, S_1 be the global state in which the protocol terminates and S be the global state constructed. We will show that there exists a run R such that $S_0 \to S \to S_1$. Let r be the actual run the system followed while executing the snapshot protocol, and let e_i^* denote the event when p_i receives a "take snapshot" message for the first time, causing p_i to record its state. An event e_i of p_i is a prerecording event if $e \to e_i^*$; otherwise, it is a postrecording event.

Consider the subsequence $\langle e', e \rangle$ of run r where e' is a postrecording event and e a prerecording event. For contradiction, assume that $e' \to e$. There are two cases to consider:

(1) both events e' and e are from the same process. If this were the case, however, then by definition e would be a postrecording event
(2) event e' is a send event of p_i and e is the corresponding receive event of p_j. If this were the case, however, then from the protocol p_i will have sent a "take snapshot" message to p_j by the time e' is executed, and since the channel is FIFO, e will also be a postrecording event

Hence, a postrecording event cannot causally precede a prerecording event and thus any $\langle \text{postrecording}, \text{prerecording} \rangle$ event pair can be swapped. Let R be the run derived from r by swapping such pairs until all postrecording events follow the prerecording events. By protocol description, postrecording events that record local states will record them at point e'. Furthermore, the channel states that are recorded are those messages that were sent by prerecording events and received by postrecording events. By construction, these are exactly those messages in the channel after the execution of event e', and so S is the state recorded by the snapshot protocol.

Part 2. Distributed transactions

A significant difference between transactions processing in a centralized and a distributed (database) system concerns the nature of failures. In a centralized system, either the system is working and transactions are processed routinely, or the system has failed and no transaction can be processed at all. In a distributed system, however, we can have **partial failures**. Some sites may be working while others have failed. The fact that failures in distributed systems do not necessarily have the crippling effect they do in centralized ones creates opportunities for greater reliability. However, ensuring that a single logical action (**commit** or **abort**) is consistently carried out at multiple sites is complicated considerably by the prospect of partial failures.

4. Failures in a distributed system

4.1. **Site failures.** When a site experiences a system failure, processing stops abruptly. When the site recovers from a failure it first executes a recovery procedure, which brings the site to a consistent state so it can resume normal processing. In this model of failure (**fail-stop**), a site is always either working correctly or not working at all

4.2. **Communication failures.** Communication links are also subject to failures. A combination of site and link failures can disable the communication between sites. This will happen if all paths between two sites A and B contain a failed site or a broken link. This phenomenon is called a **network partition**. In general, a network partition divides up the operationa sites into two or more components, where every two sites within a component can communicate with each other, but sites in different components cannot.

4.3. **Detecting failures.** A message may be **undeliverable** because its recipient is down when the message arrives, or because its sender and recipient are in different components of a network partition. In a distributed system the message is dropped: the computer network makes no further attempt to deliver it.

Both site failures and communication failures manifest themselves as the inability of one site to exchange messages with another. That is, if site A cannot communicate with site B, it is either because B has failed or because A and B belong to different components of a partition. In general, A cannot distinguish these two cases. To find out if a site can't communicate with another usually some form of **timeout** is implemented: A sends a message to B and waits for a reply within a predetermined period of time δ called the **timeout period**.

5. Atomic commitment protocol

We'll assume that for each distributed transaction T, there is a process at every **site** where T executed. The process at T's home site is called T's **coordinator**. The remaining processes are T's **participants**. The coordinator knows the names of all the participants, so it can send them messages. The participants know the name of the coordinator, but they don't necessarily know each other.

Roughly speaking, an **atomic commitment protocol** (ACP) is an algorithm for the coordinator and participants such that either the coordinator and all participants commit the transaction or they all abort it. Each process may cast exactly

one of two votes: **YES** or **NO**, and can reach exactly one of two decisions: **COMMIT** or **ABORT**. An ACP is an algorithm for processes to reach decisions such that:

Definition 5.1. (**AC1**) All processes that reach a decision reach the same one.

Definition 5.2. (**AC2**) A process cannot reverse its decision after it has reached one.

Definition 5.3. (**AC3**) The Commit decision can only be reached if all processes voted YES.

Definition 5.4. (**AC4**) If there are no failures and all processes voted YES, then the decision will be to COMMIT.

Definition 5.5. (**AC5**) Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

Condition AC1 says that the transaction terminates consistently. Note that we do not require that all processes reach a decision. We do not even require that all processes that remain operational reach a decision. However, we do require that all processes be able to reach a decision once failures are repaired (AC5). Condition AC2 says that the termination of a transaction at a site is an irrevocable decision. If a transaction commits (or aborts), it cannot be later aborted (or committed).

Condition AC3 says that a transaction cannot commit unless all sites involved in its execution agree to do so. AC4 is a weak version of the converse of AC3. A very important consequence of AC3 is that each process can unilaterally decide Abort at any time, if it has not yet voted YES. On the other hand, after voting YES a process cannot take unilateral action. The period between the moment a process votes YES and the moment it has received sufficient information to know what the decision will be is called the uncertainty period for that process. A process is called **uncertain** while it is in its uncertainty period.

5.1. **Two-phase commit protocol.** The simplest and most popular ACP is the **two-phase commit** (2PC) protocol, Assuming no failures, it goes roughly as follows:

(1) the coordinator sends a VOTE-REQ message to all participants
(2) when a participant receives a VOTE-REQ, it responds by sending to the coordinator a message containing that participant's vote: YES or NO. If the participant votes NO, it decides ABORT and stops
(3) the coordinator collects the vote messages from all participants. If all of them were YES and the coordinator's vote is also YES, then the coordinator decides and sends a COMMIT messages to all participants. Otherwise, the coordinator decides and sends an ABORT messages to all participants that voted YES (those that voted NOalready decided ABORT in step (2)). In either case, the coordinator then stops
(4) each participant that voted YES waits for a COMMIT or ABORT message from the coordinator. When it receives the message, it decides accordingly and stops

The two phases of 2PC are the voting phase (steps (1) and (2)) and the decision phase (steps (3) and (4)). A participant's uncertainty period starts when it sends a YES to the coordinator (step (2)) and ends when it receives a COMMIT or ABORT (step (4)). The coordinator has no uncertainty period since it decides as soon as it votes - with the knowledge, of course, of the participants' votes (step (3)).

It is easy to see that 2PC satisfies conditions AC1 to AC4. To satisfy AC5, first we must avoid situations in which a process is waiting for a message forever, supplying suitable timeout actions for each protocol step. Second, each process must keep some information in stable storage (**DT log**) to recover from eventual failures.

5.1.1. *Timeout actions.* There are three places in 2PC where a process is waiting for a message: in the beginning of steps (2), (3) and (4). In step (2), a participant waits for a VOTE- REQ from the coordinator. This happens before the participant has voted. Since any process can unilaterally decide ABORT before it votes YES, if a participant times out waiting for a VOTE-REQ, it can simply decide ABORT and stop. In step (3) the coordinator is waiting for YES or NO messages from all the participants. At this stage, the coordinator has not yet reached any decision. In addition, no participant can have decided COMMIT. Therefore, the coordinator can decide ABORT but must send ABORT to every participant from which it received a YES. In step (4), a participant p that voted YES is waiting for a COMMIT or ABORT from the coordinator. At this point p is uncertain, therefore the participant must consult with other processes to find out what to decide.

This consultation is carried out in a **cooperative termination protocol**: assume that participants to know each other, so they can exchange messages directly. A participant p that times out while in its uncertainty period sends a DECISION-REQ message to every other process, q, to inquire whether q either knows the decision or can unilaterahy reach one. In this scenario, p is the initiator and q a responder in the termination protocol. There are three cases:

- q has already decided COMMIT (or ABORT): q simply sends a COMMIT (or ABORT) to p, and p decides accordingly
- q has not voted yet: q can unilaterally decide ABORT. It then sends an ABORT to p, and p therefore decides ABORT
- q has voted YES but has not yet reached a decision: q is also uncertain and therefore cannot help p reach a decision

With this protocol, if p can communicate with some q for which either (1) or (2) holds, then p can reach a decision without blocking. On the other hand, if (3) holds for all processes with which p can communicate, then p is blocked. This predicament will persist until enough failures are repaired to enable p to communicate with a process q for which either (1) or (2) applies. At least one such process exists, namely, the coordinator. Thus this termination protocol satisfies AC5. However, even with the cooperative termination protocol, 2PC is subject to blocking even if only site failures occur.

5.1.2. *Recovery.* Consider a process p recovering from a failure. To satisfy AC5, p must reach a decision consistent with that reached by the other processes. If p failed before having sent YES to the coordinator (step (2) of 2PC), then p can unilaterally decide ABORT. Also, if p failed after having received a COMMIT or ABORT from the coordinator or after having unilaterally decided ABORT, then

it has already decided. In these cases, p can recover independently. However, if p failed while in its uncertainty period, then it cannot decide on its own when it recovers. Since it had voted YES, it is possible that all other processes did too, and they decided Commit while p is down. But it is also possible that some processes either voted NO or didn't vote at all and ABORT was decided.

In this case, p is in exactly the same state as if it had timed out waiting for a COMMIT or ABORT from the coordinator. Thus, p can reach a decision by using the (cooperative) termination protocol. To remember its state at the time it failed, each process must keep some information in its site's DT-log, which survives failures.

- when the coordinator sends VOTE-REQS, it writes a start-2PC record in the DT-log. This record contains the identities of the participants, and may be written before or after sending the messages
- if a participant votes YES, it writes a yes record in the DT-log, before sending YES to the coordinator. This record contains the name of the coordinator and a list of the other participants (which is provided by the coordinator in VOTE-REQ). If the participant votes NO, it writes an abort record either before or after the participant sends NO to the coordinator
- before the coordinator sends COMMIT to the participants, it writes a commit record in the DT-Iog
- when the coordinator sends ABORT to the participants, it writes an abort record in the DT-log. The record may be written before or after sending the messages
- after receiving COMMIT (or ABORT), a participant writes a commit (or ABORT) record in the DT-log

When a site S recovers from a failure, the fate of a distributed transaction executing at S can be determined by examining its DT-Iog:

- if the DT-log contains a start-2PC record, then S was the host of the coordinator. If it also contains a commit or abort record, then the coordinator had decided before the failure. If neither record is found, the coordinator can now unilaterally decide ABORT by inserting an abort record in the DT-log. For this to work, it is crucial that the coordinator first insert the commit record in the DT-log and then send COMMITS (point (3) in the preceding list)
- if the DT-log doesn't contain a start-2PC record, then S was the host of a participant. There are three cases to consider:
  - the DT-log contains a commit or abort record. Then the participant had reached its decision before the failure
  - the DT-log does not contain a yes record. Then either the participant failed before voting or voted NO, but did not write an abort record before failing (this is why the yes record must be written before YES is sent; see point (2) in the preceding list). It can therefore unilaterally ABORT by inserting an abort record in the DT-log
  - the DT-log contains a yes but no commit or abort record. Then the participant failed while in its uncertainty period. It can try to reach a decision using the termination protocol. Recall that a yes record includes the name of the coordinator and participants, which are needed for the termination protocol

Part 3. Consensus

6. The FLP impossibility result

Theorem 6.1. *In asynchronous systems, it is not possible to simultaneously guarantee safety (every non-faulty node agrees on a common value) and liveness (the algorithm terminates with a decision) for even one crash-failure, in a deterministic fashion.*

In practice, this means that algorithms for asynchronous systems sacrifice liveness for safety (2PC and Paxos), or relax the initial synchronicity assumptions (PBFT). A third option instead is the implementation of a failure detector.

Proof. (**sketch**) Assume an asynchronous system where each node starts with either (i) all zeros or (ii) all ones, and must reach a consensus (on zero or one) by running some algorithm. We can devise new configurations by flipping these "bits". Now suppose we have a configuration in which the final result depends upon the value of a crash-faulty process. It is easy to see that we can get different results whether or not this process fails. □

7. Paxos

Paxos can be described in terms of three agent roles: **proposers** that can propose values for consensus, **acceptors** that can accept a value among those proposed, and **learners** that record the chosen value. An agent can take on multiple roles: in a typical configuration, all agents play all roles. Paxos is safe for any number of crash failures, and can make progress with up to f crash failures, given $2f + 1$ acceptors. The basic Paxos protocol follow these steps:

(1) **prepare**. A proposer chooses one of the rounds associated to itself, say round i, and starts off the round by sending a $<$ PREPARE$, i >$ message to all the acceptors
(2) **promise**. When an acceptor receives the $\langle$PREPARE$, i\rangle$ message for round i, it sends a $\langle$PROMISE$, i, lrnd, lval\rangle$ message back to the proposer. In this way, the acceptor promises that it will not participate in any round smaller than i and it will stick to this promise. Along with the promise, the acceptor sends the last value it has voted $lval$ and the associated round $lrnd$
(3) **accept**. After collecting a quorum of $n - f$ promises for round i from the acceptors, the proposer sends an $\langle$ACCEPT$, v\rangle$ message to all the acceptors asking to vote for a value v selected as follow:
    (a) a value x proposed by the proposer, if no acceptor in the quorum has ever voted
    (b) the value $lval$ in the promises that is associated with the highest round $lrnd$ otherwise
(4) **learn**. If an acceptor receives a $\langle$ACCEPT$, v\rangle$ message, and if it has not promised otherwise, it votes for the value in the message and sends a $\langle$LEARN$, i, v\rangle$ message to all the learners to let them know about the vote
(5) the value is **chosen**. If a learner receives $n - f$ $\langle$LEARN$\rangle$ messages for the same round i and the same value v from a quorum of $n - f$ acceptors, then the value is chosen

7.1. **A simple proof of safety.** To prove safety, we need to prove the following three properties:

- **CS1**. Only a proposed value may be chosen
- **CS2**. Only a single value is chosen
- **CS3**. Only a chosen value may be learned by a correct learner

Property CS1 is very easy to check, acceptors only vote for values that have been proposed by the proposers. Property CS3 also is very easy to check, as learners learn a value only if it has been voted by a quorum of acceptors, the same quorum needed to chose the value. To prove CS2, it is more easy to handle the following property:

- **CS**. If acceptor a has voted for value v at round i, then no value $v' \neq v$ can be chosen in any previous round

Theorem 7.1. *In Paxos, if acceptor a has voted for value v at round i, then no value $v' \neq v$ can be chosen in any previous round.*

Proof. We prove Property CS by induction on round i. The base case, when $i = 0$, is trivially true. We now assume that the property holds for rounds $0, \ldots, i-1$ (inductive hypothesis) and we prove the property for round i. Assume A is the set of acceptors, $Q \subseteq A$ is the set of acceptors which sent their $<$ PROMISE$, i, lrnd, lval >$ and $\mid Q \mid = n - f$ is a quorum, $j < i$ is the largest $vrnd$ in the promises collected from the acceptors in Q.

First, no value can be chosen in rounds $j+1, \ldots, i-1$. Indeed, the acceptors in Q have promised to not vote in these rounds and the remaining $\mid Q \setminus A \mid$ acceptors are not enough to form a quorum. If $j = -1$, then we are done with the inductive step and with the proof. Assume acceptor a has voted for value v at round i. This is possible only if some acceptor in Q has voted value v in round $j \geq 0$. We can deduce two consequences: no value $v' \neq v$ can be chosen in round j; and, no value $v' \neq v$ can be chosen in rounds $0, \ldots, j-1$. □

Therefore we know that Paxos is safe.

7.2. **Liveness of Paxos.** In Paxos, progress is not guaranteed even if the number of failures is at most $f = \lfloor (n-1)/2 \rfloor$. Indeed, if more than one proposer starts off new rounds concurrently, then there is no guarantee that any round completes and a value is chosen. It is important to realize that we cannot do much about it - we cannot get both safety and liveness - since it is impossible to solve consensus in the presence of faults (FLP result).

To get progress we can however use a leader election protocol. One of the proposer is elected as the **coordinator**, the only one allowed to start off new rounds. In this way, no conflict occurs and, if the number of failures is at most f, the round completes with a chosen value. Of course, we are not circumventing the FLP result. In the presence of faults leader election is impossible as well. However, with Paxos we can accept that the election fails and that two or more leaders are chosen. In that case we cannot guarantee liveness but Paxos is there to guarantee safety whatsoever.

7.3. **Multi-Paxos.** Consensus protocols like Paxos are often used to get consensus on a sequence of values. A sequence of Paxos instances executes and instance s is used to agree on the s-th proposed value. A key observation in such a system is

that a single $\langle PREPARE \rangle$ message can be sent to initiate a sequence of Paxos instances. Similarly, a single $\langle PROMISE \rangle$ message can be sent to respond to the aggregate $\langle PREPARE \rangle$ message. To complete each instance, a proposer p that has a value to propose sends this value to the coordinator c. The coordinator then completes the instance by sending the proper $\langle ACCEPT \rangle$ message to the acceptors that will be followed by the $\langle LEARN \rangle$ message to the learners. Then the delay between proposing and learning in each instance consists of only three messages (down from four required for basic Paxos).

7.4. **Fast Paxos.** Further improving on multi-Paxos, **fast Paxos** is based on the following idea: We can save one message and reduce the delay between proposing and learning by allowing the proposer to send its value directly to the acceptors. To achieve this result, the coordinator can send an aggregate $\langle ACCEPT, crnd, \perp \rangle$ message (a so called **accept-any** message) to the acceptors as a response to the $\langle PROMISE \rangle$ messages. An accept-any message means that, in the same round, the acceptors can accept any value that they receive from any of the proposers. As a result, the pattern of each instance consists of only two messages, a message with the proposed value from any of the proposer to the acceptors, and a $\langle LEARN \rangle$ message from the acceptors to the learners.

The improvement of Fast Paxos in delay comes at a price. In the same round multiple proposers can send a value to the acceptors. Therefore, in the same round many different values can be voted by the acceptors. To solve this problem, in Fast Paxos we require a larger quorum of $n - f'$ acceptors, where $f' = \lfloor (n-1)/3 \rfloor$.

7.4.1. *Protocol details.* First, it is useful to understand why the quorum requirement of $n - f$, where $f = (n-1)/2$, does not work any more. Suppose that $n = 7$ and $f = 3$. During round i, the proposer responsible for round i collects a quorum of exactly $n - f = 4$ promises from the acceptors. Some of the promises have the form $\langle PROMISE, i, j, v \rangle$ (the last vote was cast in round j and the value was v) and some have the form $\langle PROMISE, i, j, v' \rangle$. This is possible in Fast Paxos since multiple values can be proposed and voted in a fast round. The problem is that the coordinator still does not know the last vote of $|A/Q| = 3$ acceptors and, unfortunately, these may be enough to form a quorum of $n - f = 4$ votes in round j either on value v or on value v'. The proposer just does not know and therefore it cannot make any safe choice in round i.

Therefore, let's assume that the proposer that is executing round i has collected $n - f'$ promises from a set Q of exactly $n - f'$ acceptors. Again, let j be the highest $lrnd$ in the promises. Moreover, let $Q_j \subseteq Q$ be the set of acceptors that last voted in round j and let $Q_j[v] \subseteq Q_j$ be the set of acceptors that last voted value v in round j. As we know, in Fast Paxos there is no guarantee that the acceptors in Q_j have last voted for the same value. Clearly, that means that we need to change the rule that Paxos uses to select the value to be sent in the Accept message. We change the rule in the following way:

- if $j = -1$ (no acceptor in Q has voted yet), select $\perp$ (start a fast round)
- if $j \geq 0$ and there exists v such that $\mid Q_j[v] \mid \geq n - 2f'$, then select v
- if $j \geq 0$ and for all v we get $\mid Q_j[v] \mid < n - 2f'$, then select any value that has been last voted in round j

7.4.2. *A simple proof of safety for Fast Paxos.* Like in Paxos, our goal is to prove the three safety Properties CS1, CS2, and CS3. Again, Property CS1 and CS3 are very easy to check and we prove Property CS as a way to prove Property CS2.

Theorem 7.2. *In Fast Paxos, if acceptor a has voted for value v at round i, then no value $v' \neq v$ can be chosen in any previous round.*

Proof. We prove Property CS by induction on round i. The base case, when $i = 0$, is trivially true. We now assume that the property holds for rounds $0, \ldots, i-1$ (inductive hypothesis) and we prove the property for round i. Let $Q \subseteq A$, $\mid Q \mid = n - f'$, be the quorum of acceptors that sent the $\langle PROMISE \rangle$ message for round i, $j < i$ be the largest $lrnd$ in the promises, $Q_j \subseteq Q$ be the set of acceptors who last voted in round j, and $Q_j[v] \subseteq Q_j$ be the set of acceptors that have last voted value v in round j. Just like in Paxos, we can easily see that no value can be chosen in rounds $j+1, \ldots, i-1$ (the acceptors in $Q \backslash A$ are not enough to form a quorum). If $j = -1$, then we are done with the inductive step and with the proof. So, let's assume that $j \geq 0$ and proceed.

Assume that acceptor a has voted for value v at round i. Then, for all $v' \neq v$ we know that $\mid Q_j[v'] \mid < n - 2f'$. As a consequence, no value $v' \neq v$ can be chosen in round j since the acceptors in $Q \backslash Q_j[v']$, which are strictly more than f', have not voted for value v' in round j. Indeed, the acceptors in $Q \backslash Q_j$ have promised not to vote any value in round j, and the acceptors in $Q_j \backslash Q_j[v']$ have not voted for value v by definition. Lastly, since at least one acceptor has voted for v in round j, no value $v' \neq v$ can be chosen in rounds $0, \ldots, j-1$ by using the inductive hypothesis on round j. □

Property CS2 easily follows from Property CS like in Paxos. Therefore, we are done and we can claim that Fast Paxos is safe.

8. Agreement problem

Agreement among the processes in a distributed system is a fundamental requirement for a wide range of applications. Many forms of coordination require the processes to exchange information to negotiate with one another and eventually reach a common understanding or agreement.

- **failure models**: among the n processes in the system, at most f processes can be faulty. Recall that in the fail-stop model, a process may crash in the middle of a step. In particular, it may send a message to only a subset of the destination set before crashing. In the Byzantine failure model, a process may behave arbitrarily
- **a/synchronous communication**: if a failure-prone process chooses to send a message to process P_i but fails, then P_i cannot detect the non-arrival of the message in an asynchronous system because this scenario is indistinguishable from the scenario in which the message takes a very long time in transit. In a synchronous system, however, the scenario in which a message has not been sent can be recognized by the intended recipient, at the end of the round
- **channel reliability**: The channels are reliable, and only the processes may fail (under one of various failure models). This is a simplifying assumption in our study

8.1. **Byzantine agreement problem.** The Byzantine agreement problem requires a designated process, called the source process, with an initial value, to reach agreement with the other processes about its initial value, subject to the following conditions:

- **agreement**. All non-faulty processes must agree on the same value
- **validity**. If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source
- **termination**. Each non-faulty process must eventually decide on a value

The validity condition rules out trivial solutions, such as one in which the agreed upon value is a constant. It also ensures that the agreed upon value is correlated with the source value. If the source process is faulty, then the correct processes can agree upon any value. It is irrelevant what the faulty processes agree upon - or whether they terminate and agree upon anything at all.

8.2. **Consensus problem.** The consensus problem differs from the Byzantine agreement problem in that each process has an initial value and all the correct processes must agree on a single value:

- **agreement**. All non-faulty processes must agree on the same (single) value
- **validity**. If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value
- **termination**. Each non-faulty process must eventually decide on a value

8.3. **Interactive consistency problem.** The interactive consistency problem differs from the Byzantine agreement problem in that each process has an initial value, and all the correct processes must agree upon a set of values, with one value for each process:

- **agreement**. All non-faulty processes must agree on the same array of values $A[v_1, \ldots, v_N]$
- **validity**. If process i is non-faulty and its initial value is v_i, then all non-faulty processes agree on v_i as the i-th element of the array A. If process j is faulty, then the non-faulty processes can agree on any value for $A[j]$
- **termination**. Each non-faulty process must eventually decide on the array A

8.4. **Equivalence.** The three problems defined above are equivalent in the sense that a solution to any one of them can be used as a solution to the other two problems. This equivalence can be shown using a reduction of each problem to the other two problems.

9. Agreement flavours

There is also a connection between the synchrony of a system and its crash tolerance. In a synchronous system, we can solve consensus for any number of failures. In an asynchronous system, consensus is impossible for even one failure.

9.1. **Agreement in a failure-free system.** In a failure-free system, consensus can be reached by collecting information from the different processes, arriving at a **decision**, and distributing this decision in the system. The decision can be reached by using an application-specific function, like the majority, max, and min functions.

9.2. **Agreement in (message-passing) synchronous systems with failures.** If a system is wholly synchronous, consensus can be solved, i.e. the trade-offs between safety and liveness can be avoided.

9.2.1. *Consensus for crash failures.* Each process has an initial value x_i. If up to f failures are to be tolerated, then the algorithm has $f+1$ rounds. In each round, a process i sends the value of its variable x_i to all other processes if that value has not been sent before. Of all the values received within the round and its own value x_i at the start of the round, the process takes the minimum, and updates x_i. After $f+1$ rounds, the local value x_i is guaranteed to be the consensus value.

- the agreement condition is satisfied because in the $f+1$ rounds, there must be at least one round in which no process failed. In this round, say round r, all the processes that have not failed so far succeed in broadcasting their values, and all these processes take the minimum of the values broadcast and received in that round. Thus, the local values at the end of the round are the same, say x_i^r for all non-failed processes. In further rounds, only this value may be sent by each process at most once, and no process i will update its value x_i^r
- the validity condition is satisfied because processes do not send fictitious values in this failure model. For all i, if the initial value is identical, then the only value sent by any process is the value that has been agreed upon as per the agreement condition
- the termination condition is seen to be satisfied

There are $f+1$ rounds, where $f < n$. The number of messages is at most $n \cdot (n-1) = O(n^2)$ in each round, hence the total number of messages is $O((f+1) \cdot n^2)$.

9.2.2. *Consensus for byzantine failures.* In a system of n processes, the Byzantine agreement problem (as also the other variants of the agreement problem) can be solved in a synchronous system only if the number of Byzantine processes f is such that $f \leq \lfloor \frac{n-1}{3} \rfloor$.

9.3. **Agreement in (message-passing) asynchronous systems with failures.** Fischer et al. showed a fundamental result on the impossibility of reaching agreement in an asynchronous (message-passing) system, even if a single process is allowed to have a crash failure. This result, popularly known as the **FLP impossibility result**, has a significant impact on the field of designing distributed algorithms in a failure-susceptible system. Observe that reaching consensus requires some form of exchange of the intial values. Hence, a running process cannot make a unilateral decision on the consensus value. The key idea of the impossibility result is that, in the face of a potential process crash, it is not possible to distinguish between a crashed process and a process or link that is extremely slow. The impossibility result is significant because it implies that all problems to which the agreement problem can be reduced are also not solvable in any asynchronous system in which crash failures may occur. Problems which reduce to the consensus problem, like the leader election, are not solvable in the face of even a single crash-failure.

10. Practical byzantine fault tolerance

Assuming an asynchronous distributed system with a byzantine failure model, we can devise an algorithm which offers both liveness and safety, provided at most $f = \lfloor \frac{n-1}{3} \rfloor$ out of a total of n nodes are simultaneously faulty. **Practical byzantine fault tollerance** does not rely on synchrony to provide safety, but it needs synchrony to provide liveness. For simplicity, we assume $n = 3f + 1$, where f is the maximum number of nodes that may be faulty. These nodes move through a succession of configurations called **views**. In a view one node is the **primary** (i.e. coordinator from Paxos) while others are **backups** (i.e. acceptors from Paxos). View changes are carried out when it appears that the primary has failed. Furthermore, we use cryptographic techniques to prevent ill actions.

11. The CAP intuition

The **CAP intuition** was introduced as a trade-off between consistency, availability, and partition tolerance:

- **consistency**: informally it means that each server returns the "right" response to each request. The meaning of consistency depends on the service. Here we discuss simple services, where its semantics are specified by a sequential specification and operations are atomic (from the PoV of the client, it is as if all operations were executed by a single centralized server)
- **availability**: each request eventually receives a response
- **partition-tolerance**: communication among the servers is not reliable, and the servers may be partitioned into multiple groups that cannot communicate with each other. here we simply treat communication as faulty

Theorem 11.1. *In a network subject to communication failures, it is impossible for any web service to implement an atomic R/W shared memory that guarantees a response to every request.*

The trade-off between consistency and availability in a partition-prone system is a particular example of the more general trade-off between safety and liveness in an unreliable system

- a **safety property** is one that states "nothing bad ever happens": it requires that at every point in every execution, the property holds. Consistency requirements are almost always safety properties
- a **liveness property** is one that states that "eventually something good happens": it says nothing about the state at any instant in time; it requires only that if an execution continues for long enough, then something desirable happens. Availability in a classic liveness property

11.1. Practical implications.

When dealing with unreliable networks, there are seemingly only two reasonable approaches: sacrifice availability or sacrifice consistency. This is the implication of the CAP theorem: we cannot achive consistency and availability in a partition-prone system.

11.1.1. *Best effort availability.*

Perhaps the most common approach to dealing with unreliable networks is to design a service that guarantees consistency regardless of the network behaviour. The service is then optimized to provide best effort availability. A recent popular example of this approach is the Chubby Lock Service used in the Google infrastructure.

11.1.2. *Best effort consistency.* For some applications, sacrificing availability is not an option: users require that it be responsive in all situations. In such situations, designers sacrifice consistency: a response is guaranteed at all times. The classic example of this is web caching, as pioneered by Akamai CDN.

12. Blockchain

The blockchain can be thought as another way of dealing with the CAP intuition.

12.1. **Bitcoin.**

12.2. **Algorand.**