Distributed Software Development
Consensus

Chris Brooks

Department of Computer Science
University of San Francisco
A fundamental problem in distributed systems is getting a set of processes or nodes to agree on one or more values.

- Is a procedure continuing or aborted?
- What value is stored in a distributed database?
- Which process is serving as coordinator?
- Has a node failed?

There are a set of related problems that require a set of processes to coordinate their states or actions.
7-1: Coordination via email

An example:

△ Two people (A and B) want to meet at dusk tomorrow evening at a local hangout.
△ Each wants to show up only if the other one will be there.
△ They can send email to each other, but email may not arrive.
△ Can either one guarantee that the other will be there?
We’ll want to distinguish what sorts of failures these algorithms can tolerate.

No failure
- Some of the algorithms we’ll see can’t tolerate a failure.

Crash failure
- This means that a node stops working and fails to respond to all messages.

Byzantine failure
- A node can exhibit arbitrary behavior.
- This makes things pretty hard for us...
How can we detect whether a failure has happened?

A simple method:
- Every $t$ seconds, each process sends an “I am alive” message to all other processes.
- Process $p$ knows that process $q$ is either unsuspected, suspected, or failed

If $p$ sees $q$’s message, it knows $q$ is alive, and sets its status to unsuspected.

What if it doesn’t receive a message?
Failure detection

- Depends on our communication model.
- Synchronous communication: if after \( d \) seconds (where \( d \) is the maximum delay in message delivery) we haven’t received a message from \( p \), \( p \) has failed.
- Asynchronous or unreliable communication: if the message is not received, we can say that \( p \) is suspected of failure.
Other problems:
- What if $d$ is fairly large?
- We can think processes are still running that have in fact crashed.

This is what’s called an *unreliable* failure detector.

It will make mistakes, but, given enough information, it may still be of use.

Can provide hints and partial information.

As we look at different algorithms, we’ll need to think about whether we can detect that a process has failed.
7-6: Multicast: a brief digression

The Coulouris chapter talks quite a bit about how to achieve different properties with multicast communication.

- Reliable multicast
- Ordered multicast
  - FIFO ordering
  - Total ordering
  - Causal ordering

The punchline: Totally ordered multicast is equivalent to the consensus problem.

Implementing one or more of these on top of IP multicast could be a cool final project.
What is multicast?

Consider that a process needs to send a message to a *group* of other processes.

It could:
- Send a point-to-point message to every other process. Inefficient, plus need to know all other processes in group.
- Broadcast to all processes in subnet. Wasteful, won’t work in wide-area network.

Multicast allows the process to do a single send. Packet is delivered to all members of the group.
Notice that multicast is a packet-oriented communication.
  △ Same send/receive semantics as UDP

A process joins a multicast group (designated by an IP address)

It then receives all messages sent to that IP address.

Groups can be closed or open.

Multicast can be effectively used to do shared whiteboards, video or audio conferencing, or to broadcast speeches or presentations.
  △ Middleware needed to provide ordering.
Mutual exclusion is a familiar problem from operating systems.

- There is some resource that is shared by several processes.
- Only one process can use the resource at a time.
- Shared file, database, communications medium

Processes request to enter their critical section, then enter, then exit.

In a centralized system, this can be negotiated with shared objects. (locks or mutexes).

Distributed systems rely only on message passing!
Our goals for mutual exclusion:

- safety: Only one process uses the resource at a time.
- liveness: everyone eventually gets a turn.
  - This implies no deadlock or starvation.
- ordering: if process $i$’s request to enter its CS happens-before (in the causal sense) process $j$’s, then process $i$ should enter first.
One solution is to use a centralized server to manage access.

To enter the critical section, a process sends a request to the server.
- If no one is in a critical section, the server returns a token. When the process exits the critical section, the token is returned.
- If someone already has the token, the request is queued.

Requests are serviced in FIFO order.
7-12: Mutual exclusion: centralized server

- If no failures occur, this ensures safety and liveness.
- Ordering is not satisfied.
- Central server provides a bottleneck and a single point of failure.
7-13: **Mutual exclusion: token ring**

- Rather than dedicating a server, processes are logically arranged in a ring.
  - This may not have anything to do with network topology; we just assign an order.
- The token is passed clockwise (for example) around the ring.
- When the token is received, a process can enter its critical section.
- If the token is not needed, it is immediately passed on.
Achieves liveness and safety, but not ordering.

Also can use a lot of bandwidth when no process needs the resource.

Also can’t handle failure.

- What if the process with the token fails?
- If we can detect failure, we can generate a new token.
- Leader election can deal with this.
Assume each process has a distinct identifier
Assume each process can keep a logical clock.
A process has three states: Released, waiting, held.
When a process $p$ wants to enter the critical section:
- $p$ sets its state to waiting
- $p$ sends a message to all other processes containing its ID and logical timestamp.
- Once all other processes respond, $p$ can enter.
7-16: Mutual exclusion: multicast

- When a message is received:
  - If state is held, queue message
  - If state is waiting and timestamp is after local timestamp, queue message.
  - Else, reply immediately.

- When exiting the critical section:
  - set state to released
  - reply to queued requests.
Example: consider $p_1$, $p_2$, $p_3$.

$p_3$ doesn’t need CS. $T(p_1) = 41$, $T(p_2) = 34$.

$p_1$ and $p_2$ request CS.

$p_3$ replies immediately to both.

When $p_2$ gets $p_1$’s request, it queues it.

$p_1$ replies to $p_2$ immediately.

Once $p_2$ exits, it replies to $p_1$. 

7-18: Mutual exclusion: multicast

- Provides liveness, safety
- Also provides ordering
  - That’s the reason for logical clocks.
- Still can’t deal with failure.
- Also scaling problems.
- Optimization: can enter the CS when a majority of replies are received.
If a failure occurs, it must first be detected.  
△ As we’ve seen, this can be difficult.

Once failure is detected, a new group can be formed and the protocol restarted.

Group formation involves a two-phase protocol. 
△ Coordinator broadcasts group change to all members.
△ Once all reply, a commit is broadcast to all members.
△ Once all members reply to the commit, a new group is formed.
How can we decide which process should play the role of server or coordinator?

We need for all processes to agree.

We can do this by means of an election.

Any process can start an election

- for example, if it notices that the coordinator fails.

We would still like safety (only one process is chosen) and liveness (the election process is guaranteed to find a winner).

- Even when more than one election is started simultaneously.
7-21: Choosing a leader

- Assume each process has an identifying value.
- Largest value will be the new leader.
  - We could use load, or uptime, or a random number.
7-22: Ring-based election algorithms

6 Assume processes are arranged in a logical ring.
6 A process starts an election by placing its identifier and value in a message and sending it to its neighbor.
When a message is received:

- If the value is greater than its own, it saves the identifier and forwards the value to its neighbor.
- Else if the receiver’s value is greater and the receiver has not participated in an election already, it replaces the identifier and value with its own and forwards the message.
- Else if the receiver has already participated in an election, it discards the message.
- If a process receives its own identifier and value it knows it is elected. It then sends an elected message to its neighbor.
- When an elected message is received, it is forwarded to the next neighbor.
7-24: Ring-based election algorithms

1. Safety is guaranteed - only one value can be largest and make it all the way through the ring.
2. Liveness is guaranteed if there are no failures.
3. Inability to handle failure once again ...
7-25: Bully algorithm

The *bully* algorithm can deal with crash failures.

- Assumption: synchronous, reliable communication

When a process notices that the coordinator has failed, it sends an election message to all higher-numbered processes.

If no one replies, it declares itself coordinator and sends a new-coordinator message to all processes.

If someone replies, its job is done.

When process \( q \) receives an election message from a lower-numbered process:

- Return a reply.
- Start an election.
Guarantees safety and liveness.
Can deal with crash failures
Assumes that there is bounded message delay
  Otherwise, how can we distinguish between a crash and a long delay?
All of these algorithms are examples of the consensus problem.

- All processes must agree on a state

Let’s take a step back and think about when the consensus problem can be solved.
We’ll start with a set of processes $p_1, p_2, ..., p_n$.

All processes can propose a value, and everyone must agree at the end.

We’ll assume that communication is reliable.

Processes can fail.
- Both Byzantine and crash failures.

We’ll also specify whether processes can digitally sign messages.
- This limits the damage Byzantine failures can do.

We’ll specify whether communication is synchronous or asynchronous.
Our goals:

- Termination: eventually every process decides on a value
- Agreement: All processes agree on a value
- Integrity: If all correct processes propose the same value, that value is the one selected.
All commanders must agree whether to attack or not.

Commanders can be treacherous and send different messages each commander

- Slightly different from consensus: one process supplies a value that others must agree on.

Can this be solved? Can the head commander get consensus?

- Depends on our communication assumptions.
If we don’t have reliable communication, consensus is impossible, even without failures.

- Email example

With reliable communication, we can solve consensus for crash failures.

Multiple rounds of communication are required to account for failures.
In the Byzantine generals problem, we can solve it in a synchronous system if less than 1/3 of the processors fail.

- Three-processor problem is impossible.

Coulouris shows how to solve this - intuition is to use majority.

If processes can sign their messages, we can solve the Byzantine generals problem for any number of failures.
In asynchronous systems, the news is not so good.

In asynchronous systems, it is impossible to guarantee that we will reach consensus, even in the presence of a single crash failure.

This means that we can’t do:
- Asynchronous Byzantine generals
- Asynchronous totally ordered multicast
7-34: *What do we do in practice?*

- Notice the word *guaranteed*. We may be able to reach consensus in some cases, we just can’t promise it.
- We can introduce systems that can survive crash failures.
- We can introduce failure detectors.
- We can use randomized behavior to foil Byzantine processes.
We can survive 1/3 Byzantine failures in a synchronous system with reliable delivery.

In an asynchronous system, we can’t guarantee consensus after a single crash failure.

Without reliable communication, consensus is impossible to guarantee.

In general, we can trade off process reliability for network reliability.
Consensus can take a number of forms:
- Mutual exclusion
- Leader election
- Consensus

Many special-purpose algorithms exist.

General results about what is possible can help in designing a system or deciding how (or whether) to tackle a problem.