Distributed Agreement
Agreement Problems

- High-level goal: Processes in a distributed system reach *agreement* on a value

- Numerous problems can be cast this way
  - Transactional commit, atomic broadcast, …

- The system model is critical to how to solve the agreement problem - or whether it can be solved at all
  - Failure assumptions
  - Timing assumptions
Review: Timing / Failure Models

• Timing assumptions:
  • Synchronous – shared clock, known bounds on message delivery
  • Asynchronous – no global clock, no time bounds on message delivery
  • Partial Synchrony – clocks synchronized within some bound, timeout to manage bounds on message delivery

• Failure assumptions:
  • Fail-stop – process is correct until it stops entirely
  • Byzantine – failed process behaves arbitrarily
A rose by any other name…

- Distributed Consensus has many names (depending on the assumptions and application)
  - Reliable multicast
  - Interactive consistency
  - Atomic broadcast
  - Byzantine Generals Problem

“This has resulted in a voluminous literature which, unfortunately, is not distinguished for its coherence. The differences in notation and the haphazard nature of the assumptions obfuscates the close relationship among these problems”

– Hadzilacos & Toueg, Distributed Systems.
High-level picture

- Goal: Build reliable systems in presence of faulty components
- Common approach:
  - Send request (or input) to some “f-tolerant” server
  - Have multiple (potentially faulty) components compute same function
  - Perform majority vote on outputs to get the “correct” result

\[
\text{majority}(v_1, v_2, v_3)
\]

\(f\) faulty, \(f+1\) good components \(\Rightarrow\) 2f+1 total
Setup of Distributed Consensus

- N processes have to agree on a single value.
  - e.g.,
    - Performing a commit in a replicated/distributed database.
    - Collecting multiple sensor readings and deciding on an action.

- Each process begins with a value.
- Each process can irrevocably decide on a value.
- Up to \( f < N \) processes may be faulty.
  - How do you reach consensus if no failures?
Properties of Distributed Consensus

• **Agreement**
  • If any correct process believes that $V$ is the consensus value, then all correct processes believe $V$ is the consensus value.

• **Validity**
  • If $V$ is the consensus value, then some process proposed $V$.

• **Termination**
  • Each process decides some value $V$.

• Which of these are **Safety** properties and which are **Liveness** properties?
Fail-Stop Faults: Problem Description

- Assumptions:
  - N processes connected by a full graph
  - Each process starts with an initial value \{0,1\}
  - Synchronous setting: solution is required within a fixed number of rounds of message exchanges
  - The number of Fail-Stop faults is bounded in advance to f. A process may fail in the middle of a message sending at some round. Once a process fails, it never recovers.
  - No omission failures.
Fail-Stop Faults: Problem Requirements

- **Agreement**: all correct processes decide on the same value
- **Validity**: If a correct process decides on a value, there was a process that started with that value
Synchronous Fail-stop Consensus Algorithm

- Each process maintains a vector containing a value for each process
- In each round:
  - Send your vector to all processes
  - Update local vector according to received vectors
- After f+1 rounds, decide according to local vector
  - e.g., If you have majority 1 in the vector => decide 1; otherwise => decide 0.
- Called “Flood Set algorithm”
Synchronous Fail-stop Consensus Algorithm

- “Flood Set algorithm” run at each process $i$
  - Remember, we want to tolerate up to $f$ failures

```plaintext
Si ← \{initial value\}
for k = 1 to f+1
  send Si to all processes
  receive S_j from all j \neq i
  Si ← Si U S_j (for all j)
end for
Decide(S_i)
```

- S is a set of values
- Decide(x) can technically be various functions
  - E.g. min(x), max(x), majority(x), or some default
- Assumes nodes are connected and links do not fail!
Analysis of FloodSet

- Requires \( f+1 \) rounds because process can fail at any time, in particular, during send
  - Must guarantee 1 round in which no failure occurs

- **Agreement:** Since at most \( f \) failures, then after \( f+1 \) rounds all correct processes will evaluate \( \text{Decide}(S_i) \) the same.

- **Validity:** \( \text{Decide()} \) results in a proposed value (or default value)

- **Termination:** After \( f+1 \) rounds the algorithm completes
Example with $f = 1$, \texttt{Decide()} = \texttt{min()}

$S_1 = \{0\}$

$S_2 = \{1\}$

$S_3 = \{1\}$
Synchronous/Byzantine Consensus

• Faulty processes can behave arbitrarily
  • May actively try to trick other processes
• Algorithm described by Lamport, Shostak, & Pease in terms of Byzantine generals agreeing whether to attack or retreat.
• The generals must have an algorithm to guarantee that:
  • A. All loyal generals decide on the same plan of action
    • Implies that all loyal generals must obtain the same information
  • B. A small number of traitors cannot cause the loyal generals to adopt a bad plan
• Decide() in this case is a majority vote, default action is “Retreat”
Byzantine Generals

• Use $v(i)$ to denote value sent by $i^{th}$ general
• A traitor could send different values to different generals, so can’t use $v(i)$ obtained from $i$ directly. New conditions:
  • Any two loyal generals use the same value $v(i)$, regardless of whether $i$ is loyal or not
  • If the $i^{th}$ general is loyal, then the value that he sends must be used by every loyal general as the value of $v(i)$.
• Re-phrase original problem as **reliable broadcast:**
  • General must send an order (“Use $v$ as my value”) to lieutenants
  • Each process takes a turn as a Commanding General, sending its value to the others as Lieutenants
  • After all values are reliably exchanged, Decide()
Synchronous Byzantine Model

**Theorem**: There is no algorithm to solve consensus if only oral messages are used, unless *more than two thirds* of the generals are loyal.

- In other words, impossible if \( n \leq 3f \) for \( n \) processes, \( f \) of which are faulty
- *Oral messages* are under control of the sender
  - sender can alter a message that it received before forwarding it
- Let’s look at examples for special case of \( n=3, f=1 \)
Case 1

- Traitor lieutenant tries to foil consensus by refusing to participate

"white hats" == loyal or "good guys"
"black hats" == traitor or "bad guys"

Round 1: Commanding General sends "Retreat"

Round 2: L3 sends "Retreat" to L2, but L2 sends nothing

Decide: L3 decides "Retreat"

Commanding General 1

Loyal lieutenant obeys loyal commander. (good)

Lieutenant 2

Lieutenant 3 decides to retreat
Case 2a

- Traitor lieutenant tries to foil consensus by lying about order sent by general

**Round 1:** Commanding General sends “Retreat”

**Round 2:** L3 sends “Retreat” to L2; L2 sends “Attack” to L3

**Decide:** L3 decides “Retreat”

Loyal lieutenant obeys loyal commander. (good)

Lieutenant 2

decides to retreat
Case 2b

- Traitor lieutenant tries to foil consensus by lying about order sent by general

Round 1: Commanding General sends “Attack”

Round 2: L3 sends “Attack” to L2; L2 sends “Retreat” to L3

Decide: L3 decides “Retreat”

Loyal lieutenant disobeys loyal commander. (bad)

Decides to retreat
Case 3

- Traitor General tries to foil consensus by sending different orders to loyal lieutenants

**Round 1:** General sends “Attack” to L2 and “Retreat” to L3

**Round 2:** L3 sends “Retreat” to L2; L2 sends “Attack” to L3

Decide: L2 decides “Attack” and L3 decides “Retreat”

Loyal lieutenants obey commander. (good?) Decide differently (bad)

Lieutenant 2 decides to attack

Lieutenant 3 decides to retreat
Byzantine Consensus: \( n > 3f \)

- Oral Messages algorithm, OM\((f)\)
- Consists of \(f+1\) “phases”
- Algorithm OM\((0)\) is the “base case” (no faults)
  1) Commander sends his value to every lieutenant
  2) Each lieutenant uses value received from commander, or default
     “retreat” if no value was received
- Recursive algorithm handles up to \( f \) faults
OM(f): Recursive Algorithm

f+1 rounds:

1) OM(f): Commander sends his value to every lieutenant

2) For each lieutenant \( i \), let \( v_i \) be the value \( i \) received from commander, or “retreat” if no value was received. Lieutenant \( i \) acts as commander in Alg. OM(f-1) to send \( v_i \) to each of the \( n-2 \) other lieutenants.

3) For each \( i \), and each \( j \neq i \), let \( v_j \) be the value Lieutenant \( i \) received from Lieutenant \( j \) in step (2) (using Alg. OM(f-1)), or else “retreat” if no such value was received. Lieutenant \( i \) uses the value \( \text{majority}(v_1, \ldots, v_{n-1}) \).

4) Continue until OM(0).
Example: $f = 1, n = 4$

- Loyal commander, 1 traitor lieutenant

**Step 1:** Commander sends same value, $v$, to all

**Step 2:** Each of L2, L3, L4 executes OM(0) as commander, but L2 sends arbitrary values

**Step 3:** Decide L3 has $\{v, v, x\}$, L4 has $\{v, v, y\}$, Both choose $v$. 
Example: $f = 1$, $n = 4$

- Traitor commander, all lieutenants loyal

**Step 1:** Commander sends different value, $x, y, z$, to each

**Step 2:** Each of L2, L3, L4 executes OM(0) as commander, sending value they received

**Step 3:** Decide
- L2 has \{x, y, z\}
- L3 has \{x, y, z\},
- L4 has \{x, y, z\},

All loyal lieutenants get same result.
Example: OM(2), f=2, n=7

• OM(2): General sends value v to all six lieutenants

• Now run OM(1) six times
  • $L_i$ takes turn as general to send value received from original general to others
  • At end of each OM(1), all lieutenants agree on the value to use for $L_i$

• Finally, OM(0): All receivers run OM(0) to exchange values
  • To verify that lieutenants tell each other the same thing
  • Msg from $L_i$ of form: “$L_0$ said $v_0$, $L_1$ said $v_1$, etc..”

• All lieutenants are now using the same set of values to reach overall decision. Let’s see how..
Example: OM(2), f=2, n=7

- Traitors: L5, L6

- Now run OM(1) six times

- All loyal lieutenants decide with maj(A, A, A, A, R, R)
  => all loyal lieutenants attack!
Example: OM(2), f=2, n=7

• Traitors: C, L6

• Now run OM(1) six times

• Decision?

A,R,A,R,A

A, R, A, R, A
Decision with Bad Commander


Problem: All loyal lieutenants do NOT choose same action
Next Step of Algorithm

- Verify that lieutenants tell each other the same thing
  - Requires rounds = f+1
  - OM(0): Msg from Li of form: “L0 said v0, L1 said v1, etc...”

- What messages does L1 receive in this example?
  - OM(2): A
  - OM(0): L1 sees:
    - 2{ 3/A, 4/R, 5/A, 6/R}
    - 3{2/R, 4/R, 5/A, 6/A}
    - 4{2/R, 3/A, 5/A, 6/R}
    - 5{2/R, 3/A, 4/R, 6/A}
    - 6{ total confusion }
  - All loyals see same messages in OM(0) from L1,2,3,4, and 5

- maj(1/A,2/R,3/A,4/R,5/A,-) => All attack

Try this with f=2, n=6!
What happens in the end?

What if 6 “played nice” and sent everyone the same value?
Problem

- Lots of messages required to handle even 1 faulty process
- Need minimum 4 processes to handle 1 fault, 7 to handle 2 faults, etc.
  - But as system gets larger, probability of a fault also increases
- Problem: Traitors can lie about what others said. => Restrict this ability!
- If we use *signed messages*, instead of oral messages, can handle f faults with f+2 processes
  - Loyal general’s signature cannot be forged => limits traitors
  - Lieutenants can’t do anything alone! => Only check for traitor commander!
  - Simple majority requirement
Signed messages: case 1

- Let $x:i$ denote the value $x$ signed by general $i$
- $v:j:i$ is value $v$ signed by $j$, and then $v:j$ signed by $i$

Round 1: Commanding General sends signed “Attack” msg

Round 2: L3: signed A:1:3 to L2; L2 sends A:1:2 to L3

Decide: L3 decides correctly

Loyal lieutenant knows what order to obey. (good!)

Note that the traitor lieutenant cannot do much!
Signed messages: case 2

- Let $x:i$ denote the value $x$ signed by general $i$
- $v:j:i$ is value $v$ signed by $j$, and then $v:j$ signed by $i$

Round 1: Traitor General sends signed “A” to L2, and “R” to L3

Both loyal lieutenants decide the same thing (good!)

Also, both lieutenants know the commander is a traitor. Why?

Round 2: L2: sends signed A:1:3 to L3; L3 sends R:1:2 to L2

Decide: Both L2 and L3 have same set of orders: \{A, R\}

Decides Choice(A, R)
Conclusions

- Problem: To implement a fault-tolerant service with coordinated replicas, must **agree on inputs**
- Byzantine failures make agreement challenging
  - Produce arbitrary output, can’t detect, collude
- Use different agreement protocol depending on assumptions
  - Oral messages: Need $3f+1$ nodes to tolerate $f$ failures
    - Difficult because traitors can lie about what others said
  - Signed messages: Need $f+2$ nodes
    - Easier because traitors can only lie about other traitors
Asynch. Distributed Consensus

• Fail-Stop/Byzantine ➔ IMPOSSIBLE!

• Fischer, Lynch and Patterson (FLP) impossibility result
  • Asynchronous assumption makes it impossible to differentiate between failed and slow processes.
  • Therefore termination (liveness) cannot be guaranteed.
  • Even if an algorithm terminates, it may violate agreement (safety).
    • A slow process may decide differently than other processes thus violating the agreement property
More Byzantine Fault Tolerance

- Castro and Liskov: Practical Byzantine Fault Tolerance
  - Uses various optimizations to combine messages, reduce total communication
  - Relies on partially synchronous assumption to guarantee liveness.
  - Therefore attacks on system can only slow it down – safety is guaranteed.
  - Assumes that an attack on liveness can be dealt with in a reasonable amount of time.
  - Suitable for wide area deployment (e.g., internet)
  - Being used in Microsoft Research’s Farsite distributed file system
- Zyzzyva: Speculative Byzantine Fault Tolerance
  - The Next 700 BFT Protocols, Guerraoui et al. (Eurosys 2010)
  - A form of BFT is used in Bitcoin!