Consensus: Example

Proposal: move midterm date to another day

Consensus needed

◦ All students must be OK with new date (input)
◦ Everyone must know the final decision (agreement)
What is Consensus?

N processes

Each process \( p \) has

- input variable \( x_p \) : initially either 0 or 1
- output variable \( y_p \) : initially \( b \) (\( b=\text{undecided} \)) – can be changed only once

Consensus problem: design a protocol so that either

- all non-faulty processes set their output variables to 0
- \( Or \) all non-faulty processes set their output variables to 1
- There is at least one initial state that leads to each outcomes 1 and 2 above
System Model!

Processes fail only by crash-stopping

*Synchronous* system: bounds on:
  ◦ Message delays
  ◦ Max time for each process step

*Asynchronous* system: no such bounds
Consensus, try 1

1. Each process $p_i$ multicasts its input $x_i$ to all other processes
2. Upon receiving $x_j$ from all processes $p_j$, set $y_i = f(x_1, \ldots, x_n)$
   - E.g., $y_i = \min(x_1, \ldots, x_n)$
Synchronous Consensus (try 2)

1. Each process $p_i$ multicasts its input $x_i$ to all other processes
2. If a process does not reply with a timeout
3. Upon receiving $x_j$ from all processes $p_j$, set $y_i = f(x_1, ..., x_n)$ (omitting any processes that timed out)
   - E.g., $y_i = \min(x_1, ..., x_n)$
Consensus Problem

System of N processes \((P_1, P_2, \ldots, P_n)\)

Each process \(P_i\):

- begins in an *undecided* state.
- proposes value \(v_i\).
- at some point during the run of a consensus algorithm, sets a decision variable \(d_i\) and enters the *decided* state.
Required Properties

**Termination:** Eventually each process sets its decision variable.

**Agreement:** The decision value of all correct processes is the same.
- If $P_i$ and $P_j$ are correct and have entered the *decided* state, then $d_i = d_j$.

**Integrity (non-triviality):** If all the correct processes proposed the same value, then any correct process in the *decided* state has chosen that value.

*or* There is some initial state and execution that leads to deciding 0 and to deciding 1
How do we agree on a value?

Ring-based leader election

- Send proposed value along with *elected* message.
- Turnaround time: $3NT$ worst case and $2NT$ best case (without failures).
- $T$ is the time taken to transmit a message on a channel.
- $O(Nft)$ if up to $f$ processes fail during the election run.

Bully algorithm

- Send proposed value along with the *coordinator* message.
- Turnaround time: $4T$ in the worst case without failures.
- More than $2fT$ if up to $f$ processes fail during the election run.

*What’s the best we can do?*
Consider the simplest algorithm

Let’s assume the system is synchronous.

Use a simple B-multicast:

◦ All processes B-multicast their proposed value to all other processes.
◦ Upon receiving all proposed values, pick the minimum.

Time taken under no failures?

◦ One message transmission time (T)

What can go wrong?

◦ If we consider process failures, is a simple B-multicast enough?
B-multicast is not enough for this

\{v_1, v_2, v_3, v_5\}

\{v_1, v_2, v_3, v_4, v_5\}

Need R-multicast
B-multicast is not enough for this

\{v_1, v_2, v_3, v_5\}

\{v_1, v_2, v_3, v_4, v_5\}

Need R-multicast
B-multicast is not enough for this

\[ \{v_1, v_2, v_3, v_4, v_5\} \]

Need R-multicast
Handling failures

- P4 fails before sending \( v_4 \) to anyone.
- What should other processes do?
  - Detect failure. *Timeout!*
  - Assume proposals are sent at time ‘s’.
  - Worst-case skew is \( \epsilon \).
  - Maximum message transfer time (including local processing) is \( T \).
- What should the timeout value be?
Handling failures

- Assume proposals are sent at time ‘s’.
- Worst-case skew is $\epsilon$.
- Maximum message transfer time (including local processing) is $T$.
- What should the timeout value be?
- Option 1: $\epsilon + T$
  - Pi waits for ($\epsilon + T$) time units after sending its proposal at time ‘s’.
  - Any other process must have sent proposed value before $s + \epsilon$.
  - The proposed value should have reached Pi by ($s + \epsilon + T$).
- Will this work?
Handling failures

- Assume proposals are sent at time ‘s’.
- Worst-case skew is $\epsilon$.
- Maximum message transfer time (including local processing) is $T$.
- What should the timeout value be?
- Option 1: $\epsilon + T$
  - Pi waits for $(\epsilon + T)$ time units after sending its proposal at time ‘s’.
  - Any other process must have sent proposed value before $s + \epsilon$.
  - The proposed value should have reached Pi by $(s + \epsilon + T)$.
- Will this work?
Handling failures

- Assume proposals are sent at time ‘s’.
- Worst-case skew is $\epsilon$.
- Maximum message transfer time (including local processing) is $T$.
- What should the timeout value be?
- How about $\epsilon + 2T$?
- Will this work?
Handling failures

\{v_1, v_2, v_3, v_5\} \quad \{v_1, v_2, v_3, v_4, v_5\}

\{v_1, v_2, v_3, v_5\}

\{v_1, v_2, v_3, v_4, v_5\}
Handling failures

- Assume proposals are sent at time ‘s’.
- Worst-case skew is $\epsilon$.
- Maximum message transfer time (including local processing) is $T$.
- What should the timeout value be?
- How about $\epsilon + 3*T$?
  - Will this work?
Handling failures

- Assume proposals are sent at time ‘s’.
- Worst-case skew is $\varepsilon$.
- Maximum message transfer time (including local processing) is $T$.
- What should the timeout value be?
- How about $\varepsilon + 3*T$?
- Will this work?
Handling failures

- Assume proposals are sent at time ‘s’.
- Worst-case skew is $\epsilon$.
- Maximum message transfer time (including local processing) is $T$.
- What should the timeout value be?
- Timeout = $\epsilon + (f+1)T$ for up to $f$ failed process.

Also holds for R-multicast from a single sender.
Round-based algorithm

- For a system with at most $f$ processes crashing
  - All processes are synchronized and operate in “rounds” of time.
    - One round of time is equivalent to $\epsilon + T$ units.
    - At each process, the $i^{th}$ round
      - starts at local time $s + (i - 1)(\epsilon + T)$
      - ends at local time $s + i(\epsilon + T)$
    - The start or end time of a round in two different processes differs by at most $\epsilon$.
  - The algorithm proceeds in $f+1$ rounds.
  - Assume communication channels are reliable.
import time

TIME_BOUND = 100  # enough time to # receive all msgs in a round

def consensus(input, max_fail):
    
    `input`: this process's input
    `max_fail`: number of process failures to tolerate

    values = { input }  # set of values # currently known to this process

    for round in range(max_fail + 1):
        round_start = time.time()
        # send values to all other procs
        multicast(values)
        while time.time() < round_start +
            TIME_BOUND:
            for message in get_messages():
                # add received values to ours
                values = values | message

    return min(values)  # or any
    # deterministic function over the # value set
Proof

**Lemma 1**: after a round with no failures, all non-faulty processes have the same values set

- **Proof**: if there are no failures, B-multicast is reliable
  - All processes receiver the same messages

**Main proof**:

- There are more rounds than expected failures
- Therefore at least one round has no failures
- After this round all non-crashed processes have the same value set
- Value set will remain the same in future rounds
Consensus in an Asynchronous System

Messages have arbitrary delay, processes arbitrarily slow

Impossible to achieve!
- even a single failed is enough to avoid the system from reaching agreement!
- a slow process indistinguishable from a crashed process

Impossibility Applies to any protocol that claims to solve consensus!

Proved in a now-famous result by Fischer, Lynch and Patterson, 1983 (FLP)
- Stopped many distributed system designers dead in their tracks
- A lot of claims of “reliability” vanished overnight
Recall

Each process $p$ has a state
- program counter, registers, stack, local variables
- input register $x_p$: initially either 0 or 1
- output register $y_p$: initially $b$ ($b=$undecided)

Consensus Problem: design a protocol so that either
- all non-faulty processes set their output variables to 0
- Or non-faulty all processes set their output variables to 1
- (No trivial solutions allowed)
States, Events, and Schedule

A *global state* consists of:
- The state of all processes
- The state of all communication channels (messages “in flight”)

An *event* involves transitions between global states
- Read one “in flight” message
- Do some processing
- Send one or more messages

A *schedule* is a sequence of events
Initial state:
P1: [] P2: [] P3: [], messages: []

P1 sends message x

P2 sends message y
P3 receives X
P3 receives Y
P1 sends message x
Global Message Buffer

send(p', m)

receive(p')

may return null

"Network"
Different Definition of “State”

State of a process

Configuration: = Global state. Collection of states, one per process; and state of the global buffer

Each Event consists atomically of three sub-steps:
- receipt of a message by a process (say p), and
- processing of message, and
- sending out of all necessary messages by p (into the global message buffer)

Note: this event is different from the Lamport events

Schedule: sequence of events
Event $e' = (p', m')$

Schedule $s = (e', e''')$

Configuration C

Equivalent
Lemma 1

Schedules are commutative

Schedule s1

Schedule s2

s1 and s2 can each be applied to C
• involve disjoint sets of receiving processes

C

C'

C''

s1

s2
State Valencies

Let config. C have a set of decision values V reachable from it

- If $|V| = 2$, config. C is bivalent
- If $|V| = 1$, config. C is said to be 0-valent or 1-valent, as is the case

Bivalent means outcome is unpredictable
conf = state of proc
msgs in flight

event = recv msg
processing
set msgs

0-valent
- all live proc have decide

1-valent
What we’ll Show

There exists an initial configuration that is bivalent

Starting from a bivalent config., there is always another bivalent config. that is reachable
Lemma 2

Some initial configuration is bivalent

• Suppose all initial configurations were either 0-valent or 1-valent.
• Place all configurations side-by-side, where adjacent configurations differ in initial $x_p$ value for *exactly one* process.
• Creates a lattice of states

1 1 0 1 0 1

• There *has* to be *some* adjacent pair of 1-valent and 0-valent configs.
Lemma 2

Some initial configuration is bivalent

• There has to be some adjacent pair of 1-valent and 0-valent configs.
• Let the process p be the one with a different state across these two configs.
• Now consider the world where process p has crashed
  Both these initial configs. are indistinguishable. But one gives a 0 decision value. The other gives a 1 decision value.
  So, both these initial configs. are bivalent
What we’ll Show

There exists an initial configuration that is bivalent

Starting from a bivalent config., there is always another bivalent config. that is reachable
Lemma 3

Starting from a bivalent config., there is always another bivalent config. that is reachable.
Lemma 3

A bivalent initial config.

Let $e=(p,m)$ be an applicable event to the initial config.

Let $C$ be the set of configs. reachable without applying $e$
Lemma 3

A bivalent initial config.

let \( e = (p, m) \) be an applicable event to the initial config.

Let \( C \) be the set of configs. reachable without applying \( e \)

Let \( D \) be the set of configs. obtained by applying single event \( e \) to any config. in \( C \)
Lemma 3

C

[bivalent]

C

[don’t apply event e=(p,m)]
Claim. Set D contains a bivalent config.

Proof. By contradiction. That is, suppose D has only 0- and 1-valent states (and no bivalent ones)

There are states D₀ and D₁ in D, and C₀ and C₁ in C such that
- D₀ is 0-valent, D₁ is 1-valent
- D₀ = C₀ followed by e = (p,m)
- D₁ = C₁ followed by e = (p,m)
- And C₁ = C₀ followed by some event e’ = (p’,m’)
  (why?)
Proof. (contd.)

• Case I: $p'$ is not $p$

• Case II: $p'$ same as $p$

Why? (Lemma 1)
But $D_0$ is then bivalent!
Proof. (contd.)

• Case I: $p'$ is not $p$

• Case II: $p'$ same as $p$

But $A$ is then bivalent!
Lemma 3

Starting from a bivalent config., there is always another bivalent config. that is reachable.
Putting it all Together

Lemma 2: There exists an initial configuration that is bivalent

Lemma 3: Starting from a bivalent config., there is always another bivalent config. that is reachable

Theorem (Impossibility of Consensus): There is always a run of events in an asynchronous distributed system (given any algorithm) such that the group of processes never reaches consensus (i.e., always stays bivalent)
  ◦ “The devil’s advocate always has a way out”
Why is Consensus Important? –

Many problems in distributed systems are equivalent to (or harder than) consensus!

- Agreement, e.g., on an integer (harder than consensus, since it can be used to solve consensus) is impossible!
- Leader election is impossible!
  - A leader election algorithm can be designed using a given consensus algorithm as a black box
  - A consensus protocol can be designed using a given leader election algorithm as a black box
- Accurate Failure Detection is impossible!
  - Should I mark a process that has not responded for the last 60 seconds as failed? (It might just be very, very, slow)
  - Completeness + Accuracy impossible to guarantee
Summary

Consensus Problem

- agreement in distributed systems
- Solution exists in synchronous system model (e.g., supercomputer)
- Impossible to solve in an asynchronous system (e.g., Internet, Web)
  - Key idea: with only one process failure and arbitrarily slow processes, there are always sequences of events for the system to decide any which way. Regardless of which consensus algorithm is running underneath.
- FLP impossibility proof