Consensus

Material derived from slides by I. Gupta, M. Harandi, J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya
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 $b$ ($b$=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 non-faulty all processes set their output variables to 1
- There is at least one initial state that leads to each outcomes 1 and 2 above
Solve Consensus!

Uh, what’s the model? (assumptions!)

Processes fail only by crash-stopping

Synchronous system: bounds on
- Message delays
- Max time for each process step
- e.g., multiprocessor (common clock across processors)

Asynchronous system: no such bounds!
- e.g., The Internet! The Web!
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 $y_i = f(x_1, ..., x_n)$
   - E.g., $y_i = \min(x_1, ..., x_n)$
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., $y_i = \min(x_1, ..., x_n)$

\[\begin{align*}
@ p_1 & \quad \min(0, y_1, 1) = 0 \\
@ p_3 & \quad \min(-, y_1, 1) = 1
\end{align*}\]
Consensus in Synchronous Systems

For a system with at most f processes crashing, the algorithm proceeds in f+1 rounds (with timeout), using basic multicast (B-multicast).

Values$r_i$: the set of proposed values known to process $p=P_i$ at the beginning of round $r$.

Initially $Values^0_i = \{\}$; $Values^1_i = \{v_i=x_p\}$

for round $r = 1$ to $f+1$ do
  multicast ($Values^r_i$)
  $Values^{r+1}_i \leftarrow Values^r_i$
  for each $V_j$ received
  $Values^{r+1}_i = Values^{r+1}_i \cup V_j$
  end
  end

$y_p = d_i = \min(Values^{f+1}_i)$
Why does the Algorithm Work?

Proof by contradiction.

Assume that two non-faulty processes differ in their final set of values. Suppose $p_i$ and $p_j$ are these processes.

Assume that $p_i$ possesses a value $v$ that $p_j$ does not possess.

- In the last round, some third process, $p_k$, sent $v$ to $p_i$, and crashed before sending $v$ to $p_j$.
- Any process sending $v$ in the penultimate round must have crashed; otherwise, both $p_i$ and $p_j$ should have received $v$.
- Proceeding in this way, we infer at least one crash in each of the preceding rounds.
- But we have assumed at most $f$ crashes can occur and there are $f+1$ rounds $\implies$ contradiction.
N-1 rounds
- \[ \text{either} \leq N-2 \text{ crashes} \implies \text{1 "crash-free round"} \]
- \[ N-1 \text{ crashes} \implies 1 \text{ process left} \]
- \[ N \text{ crashes} \implies 0 \text{ processes left} \]

\[ p_1 \quad p_2 \quad p_3 \quad p_n \]

\[ r \]

\[ r \]

\[ \text{failure} \]

\[ v_i \]

\[ x \]

\[ y \]

\[ \text{ protocol } \]

\[ \text{ protocol } \]

\[ \text{ protocol } \]

\[ \text{ protocol } \]
R-Multicast

1. Each process $p_i$ **R-multicasts** its input $x_i$ to all other processes

2. Upon receiving $x_i$ from all processes $p_j$, set $y_i = f(x_1, ..., x_n)$

Values $^{0.5}_{i=1} =$ \{ \}

Values $^{1.5}_{i=1} =$ \{ $v_i = x_p$ \}

for round $r = 1$ to $f+1$ do

multicast (Values $^r_{i=1}$)

Values $^{r+1}_{i=1}$ ←

for each $V_j$ received

Values $^{r+1}_{i=1} \cup V_j$

end

end

$y_p = d_i = \text{minimum}(\text{Values}^{f+1}_{i=1})$

Sync - (consensus)
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)
Global Message Buffer

send(p', m)

receive(p') may return null

"Network"

p

p'
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 $e' = (p', m')$

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''

s2

s1

\[ m^', m^" \]

\[ p' \text{ rec } u' \]

\[ p'' \text{ rec } u'' \]

\[ \text{ sends } \overline{u} \]
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

[Diagram showing different valences and their implications]
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

\[
\begin{array}{cccccc}
1 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

- 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
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

[diagram with text: 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 foll. by e=(p,m)
- D1=C1 foll. 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