CS505: Distributed Systems

Lecture 10: Consensus
Consensus impossibility result

Consensus with $\diamond S$

Consensus with $\Omega$
Consensus

*Most famous problem in distributed computing*

*Intuitively: a group of processes need to reach agreement on a common value*
  - E.g., total order broadcast: next message(s) to deliver

*Still “light form” of agreement; “harder” problems exist, e.g.,*
  - Non-blocking atomic commit (for distributed transactions)
  - Leader election (for passive replication)
  - Mutual exclusion (for distributed critical sections)
Definition of (Binary) Consensus

- Defined by two primitives
  - propose
  - decide

- Processes propose either 0 or 1

- Processes decide on same value
## Properties

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I.</strong></td>
<td><strong>Validity</strong></td>
</tr>
<tr>
<td></td>
<td>Any value decided is a value proposed</td>
</tr>
<tr>
<td><strong>II.</strong></td>
<td><strong>Integrity</strong></td>
</tr>
<tr>
<td></td>
<td>No process decides twice</td>
</tr>
<tr>
<td><strong>III.</strong></td>
<td><strong>Agreement</strong></td>
</tr>
<tr>
<td></td>
<td>No two correct processes decide differently</td>
</tr>
<tr>
<td><strong>IV.</strong></td>
<td><strong>Termination</strong></td>
</tr>
<tr>
<td></td>
<td>Every correct process eventually decides</td>
</tr>
</tbody>
</table>
[FLP’87] states there is no deterministic solution to consensus in asynchronous systems if even one single process can fail.

- Any fault-tolerant algorithm solving consensus has runs that never terminate.
- These runs can be very unlikely (“probability zero”).
- Yet they imply that we can’t find a totally correct solution.
- And so “consensus is impossible” (“not always possible”).
Execution, Configuration, Events

- Set of processes $p_i$, each process with a state

- Configuration
  - Set of states of all processes at some moment

- Events
  - send and receive
  - Can change the state at a process
  - In particular latter can be delayed locally

- Execution
  - Sequence of configuration and events; schedules
Configuration $C$

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

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

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

Equivalent
Commutative Schedules

Schedule s1 and s2 involve disjoint sets of receiving processes.
We can classify configurations as

- **Bivalent**
  - Decision is not already (pre-)determined, outcome is unpredictable; can be 0 or 1

- **Univalent**
  - 0 - *valent*, will result in deciding 0
  - 1 - *valent*, will result in deciding 1
Proof Sketch

- The goal is to construct an execution that does not decide, showing that the protocol remains forever indecisive.

- Start with an initially bivalent state, identify an execution that would lead to a univalent state, let’s say 0-valent.

- The switch from bivalent to univalent is due to an event $e = (p, m)$ in which some process $p$ receives some message $m$. 
We will delay the \( e \) event for a while
- Delivery of \( m \) would make the run univalent but \( m \) is delayed (fair-game in an asynchronous system)

Since the protocol is indeed fault-tolerant (\( p \) can fail) there must be a run that leads to the other univalent state
- \( e \) can be “lost” if \( p \) fails
- Since the configuration is bivalent, both outcomes must still be possible

Now let \( m \) be delivered, this will bring the system back in a bivalent state
**Proof: More Details**

- **Lemma 1**: There exists an initial configuration that is bivalent.
- **Lemma 2**: Starting from a bivalent configuration $C$ and an event $e = (p, m)$ applicable to $C$, consider $C$ the set of all configurations reachable from $C$ without applying $e$ and $D$ the set of all configurations obtained by applying $e$ to the configurations from $C$, then $D$ contains a bivalent configuration.

**Theorem**: There is always a sequence of events in an asynchronous distributed system such that the group of processes never reach consensus.
Lemma 1: Proof Sketch

Lemma 1: There exists an initial configuration that is bivalent.

Assume by contradiction that there is no bivalent initial configuration. List all initial configurations. There must be both 0-valent and 1-valent initial configurations. (Why?)

Consider a 0-valent initial configuration $C_0$ adjacent to a 1-valent configuration $C_1$: they differ only in the value corresponding to process $p$. 
Lemma 1: There exists an initial configuration that is bivalent

Let this process $p$ crash

Note that both $C_0$ and $C_1$ will lead to the same final configuration with the exception of internal state of $p$ (they were identical, the only difference was determined by $p$)

If decision reached is 1, then $C_0$ must be bivalent, if decision is 0 then $C_1$ must be bivalent

*Thus, there exists an initial configuration that is bivalent*
Lemma 2

A bivalent config.

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

Let $C$ be the set of configs. reachable without applying $e$.
Lemma 2 (2)

A bivalent config.

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

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

Let \( \mathcal{D} \) be the set of configs. obtained by applying \( e \) to a config. in \( C \).
Claim. $\mathcal{D}$ contains a bivalent config.

Proof. By contradiction. $\Rightarrow$ assume there is no bivalent config in $\mathcal{D}$

$\triangleright$ There are adjacent configs. $C_0$ and $C_1$ in $C$ such that

$C_1 = C_0$ followed by $e'$

$\triangleright$ and

- $e' = (p', m')$
- $D_0 = C_0$ followed by $e = (p, m)$
- $D_1 = C_1$ followed by $e = (p, m)$
- $D_0$ is 0-valent, $D_1$ is 1-valent (if not, $C_0$ would be univalent $\Rightarrow D$)

$i$-valent config $E_i$ reachable from $C$ exists (because $C$ is bivalent)

- If $E_i$ in $C$, then $F_i = e(E_i)$
- Else $e$ was applied reaching $E_i$

Either way there exists $F_i$ in $\mathcal{D}$ for $i = 0$ and 1 both
Proof. (contd.)

- Case I: $p'$ is not $p$
- Case II: $p'$ same as $p$
Proof. (contd.)

- Case I: $p'$ is not $p$

- Case II: $p'$ same as $p$

But $A$ is then bivalent!
Consensus with ◊S

Rotating coordinator paradigm [CT’96]
- Algorithm proceeds in “unsynchronized” rounds
- Requires majority of correct processes
  - ◊S only eventually (weakly) accurate
  - False suspicions could lead to two or more subsets of \( \prod \) with different decisions

Early termination [MR’99]
For Process $p_i$

upon propose(v):
  $r \leftarrow 0$ // current round
  while not decided do
    $c \leftarrow (r \text{ mod } n) + 1$ // current coordinator
    $u \leftarrow \bot$ // value received from coordinator $p_c$ or $\bot$ if none
    if $i = c$ then send(PROPOSE, $r$, v) to all
    wait for receive(PROPOSE, $r$, v') from $p_c$ or $p_c \in D_i$
    if (PROPOSE, $r$, v') was received then $u \leftarrow v'$
    send(VOTE, $r$, u) to all
    wait for receive(VOTE, $r$, u') from majority of processes
    $U \leftarrow$ set of values received in VOTE messages
    if $U = \{u'\}$ for some $u' \neq \bot$ then send(DECIDE, u') to all
    else if $U = \{u', \bot\}$ then v $\leftarrow u'$
    $r \leftarrow r + 1$

upon receive(DECIDE, v):
  if (not decided)
    decided $\leftarrow$ true
    send(DECIDE, v) to all
  decide(v)
Agreement

1. If two servers decide in the same round, then they decide the same value.

2. Suppose some server decides $v'$ in round $r$. Then the value $v'$ is contained in the propose message of round $r$ and has been “locked” in the sense that it is not possible for any server in round $r' > r$ to decide $u' \neq v'$ or to assign $u' \neq v'$ to its $v$ because every two sets of a majority of processes intersect.
1. If some server decides, then every other correct server eventually decides
   - Cf. Reliable Broadcast, and 2.

2. There is some round in which the coordinator $p_c$ is not suspected by any server
   - By the failure detector properties
   - All correct servers decide in this round
Discussion

- How about uniformity?
  - V. Uniform Agreement
    No two (correct or faulty) processes decide differently

- Fairness?

- Reliable Broadcast?
Leader-based consensus [MR’01]
- Uses leader oracle

Idea
- Rotating coordinator “tries to look for a leader”
- If oracle is a corresponding abstraction solution can be more effective
For Process $p_i$

upon propose(v):
  $r \leftarrow 0$  // current round
  $u \leftarrow v$  // current estimate
  while not decided do
      $r \leftarrow r + 1$
      send(PHASE1, r, u) to all  // phase 1
      wait for (receive(PHASE1, r, v') from $p_l$ s.t. $l$=leader$_i$)
      $u \leftarrow v'$
      send(PHASE2, r, u) to all  // phase 2
      wait for (receive(PHASE2, r, u') from majority of processes)
      $U \leftarrow$ set of values $u'$ received in vote messages
      if $U = \{u\}$ for some $u' \neq \bot$ then aux $\leftarrow u'$
      else aux $\leftarrow \bot$
      send(PHASE3, r, aux) to all  // phase 3
      wait for (receive(PHASE3, r, aux') from majority of processes)
      if (received (PHASE3, r, aux’) with aux’ = v’ $\neq$ ⊥) then u $\leftarrow$ v'
      if (all (PHASE3, r, aux’) messages are such that aux’ $\neq$ ⊥) then
          broadcast(DECIDE, u); decided $\leftarrow$ true
  upon deliver(DECIDE, v):
      decided $\leftarrow$ true
      decide(v)
1. No correct process blocks forever in a round
   - Phase 1: since there is eventually a correct leader
   - Phase 2 and 3: every process sends to all, so majority is received

2. Every correct process decides
   - Eventual leader and 1. imply that there is round $r$ such that a correct process is leader and is seen by every correct process
   - Phase 1: all get estimate of leader
   - Phase 2 and 3: they exchange that estimate, and thus decide
Agreement

No two processes decide different values

- A process decides \( v \) during \( r \)
- At the end of phase two, \( aux \) must have been \( v \) or \( \perp \) for any process (there can only be one majority)
- A process only decides if it received same \( aux \neq \perp \) from majority; since sets overlap, at least one such message was received by other (non-faulty at that time) processes, which thus updated its \( u \)
Performance

[UHSK’04]

- Paxos (leader-based) outperforms rotating coordinator (four phases [CT’96]) with at least one crash
- Also with large number of processes and no crashes


References


