Consensus and related problems

Today
- Coordination and agreement in group communication
- Consensus

*How can a set of processes agree on the order of delivery for some messages?*

*How can they agree on some value or common decision, no matter the domain?*
Group communication – system model

- A collection of communicating processes
- Processes may be in multiple groups
- A $multicast(m,g)$ sends msg $m$ to all processes in a group $g$
- $Deliver(m)$ delivers a msg sent to the calling process
- Every msg carries the uid of the sender and the group destination
Basic multicast

- A correct (non-failing) process will eventually deliver the msg, as long as the multicaster does not crash.

- An algorithm
  - $B$-multicast($g, m$): for each process $p$ in $g$, send($p, m$)
  - On receive($m$) at $p$, $B$-deliver($m$) at $p$

  - send($p, m$) is a reliable one-to-one send.

- Potential ack-implosion
  - Acks from reliable send will probably arrive close to each other, …
  - overloaded multicast process will start to drops them, …
  - leading to more msgs and more acks …
Reliable multicast

- Must satisfy
  - Integrity – a correct $p$ in group $g$ delivers $m$ at most once
  - Validity – if a correct process multicast $m$, then it will eventually deliver $m$
  - Agreement – if a correct process delivers $m$, then all other correct processes in group will eventually deliver $m$

- Building on B-multicast

  **R-multicast – for each $p$ on $g$, B-multicast($g,m$)**

  **On B-deliver at process $q$**

  1. If $m$ is not received
  2. $\text{Received} = \text{Received} \cup \{m\}$
  3. If ($q \neq p$) then B-multicast($g,m$)
  4. R-deliver $m$

  **End**
Reliable multicast and failures

- Agreement so far refers only to *correct* processes – those that never fail
  - Uniform property – holds whether or not processes are correct

- Uniform agreement
  - if a process, correct or failed, delivers message \( m \), then all correct processes in \( g \) eventually deliver \( m \)
  - Matters if a process can take an action that produces an observable inconsistency before it crashes
  - Consider a “minor” change in the previous code …

---

**On B-deliver at process \( q \)**

If \( m \) is not received

\[
\text{Received} = \text{Received} \cup \{m\}
\]

- \( R\)-deliver \( m \)
  - If \( (q \neq p) \) then B-multicast(\( g, m \))

End

Crash!
Ordering multicast

- Basic algorithm delivers messages in arbitrary order
- Common ordering requirements
  - FIFO – if a correct process issues \( \text{multicast}(g,m) \), \( \text{multicast}(g,m') \), every correct process that delivers \( m' \), delivers \( m \) before \( m' \)
  - Causal – if \( \text{multicast}(g,m) \rightarrow \text{multicast}(g,m') \) based only on messages exchanged by processes in \( g \), every correct process that delivers \( m' \), delivers \( m \) before \( m' \)
  - Total – if a correct process delivers \( m \) before \( m' \), every correct process that delivers \( m' \), delivers \( m \) before \( m' \)

*How would you implement them?*
Causally ordered multicasting

- Ensure that message $m$ is delivered only if all causally preceding messages have already been delivered

- Clock adjustment only when sending/receiving
  - Consider two processes, $p_i$ sending a message to $p_j$
  - $p_i$ increments $V_{ij}[i]$ only when sending a message
  - $p_j$ “adjusts” $V_j$ when receiving a message

- $p_j$ postpones delivery of $m$ until:
  - $ts(m)[i] = V_{ij}[i] + 1$
    - $m$ is next msg $p_j$ was expecting from $p_i$
  - $ts(m)[k] \leq V_{ij}[k]$ for $k \neq j$
    - $p_j$ has seen all msgs seen by $p_i$ when it sent the message
Causally ordered multicasting

- Suppose $p_j$ receives $m$ from $p_i$ with timestamp $ts(m)$
- $P_j$ postpones delivery of $m$ until:
  - $ts(m)[i] = VC_j[i] + 1$
  - $ts(m)[k] \leq VC_j[k]$ for $k \neq j$
Example use – totally ordered multicast

- To guarantee that concurrent updates on a replicated database are seen in the same order everywhere
  - P1 adds $100 to an account (initial value: $1000)
  - P2 increments account by 1%
  - There are two replicas

In absence of proper synchronization: replica #1 ← $1111, while replica #2 ← $1110
Ordering multicast

- Note that
  - FIFO and Causal are partial orderings
  - Causal => FIFO
  - Total ordering allows message delivery to be ordered arbitrarily, as long as it is the same for all
  - No mention of reliability (if you don’t deliver \( m' \), you are OK)

- We can define some hybrids
  - FIFO-total, causal-total, reliable FIFO, reliable causal, a reliable totally ordered multicast (sometimes called ‘atomic multicast’)


Consensus and failures

- How to make process agree on a value after one or more have proposed what the value should be?
- They should reach consensus even if some processes may fail
  - Failure – System cannot meet its promises
  - Error – Part of system’s state that can lead to a failure

- Many ways to fail …
Failure models

- **Failures**
  - Crash failures – a component simply halts, but behaves correctly before halting
  - Omission failures – … fails to respond to incoming requests
  - Timing failures – output is correct, but lies outside a specified real-time interval
  - Response failures – output is incorrect
  - Arbitrary/byzantine failures – may produce arbitrary output and be subject to arbitrary timing failures

- **Basic approach to masking faults – redundancy**
  - Information – add extra bits to a message for reconstruction
  - Time – do something more than once if needed
  - Physical – add copies of software/hardware
Consensus – definition

• To reach consensus
  – Every process \( p_i \) begins in the \textit{undecided} state and proposes a value \( v_i \)
  – Processes exchange values
  – Each sets the value of a decision variable \( d_i \), entering the \textit{decided} state

• Requirements
  – Termination – Eventually all correct processes set their decision variable
  – Agreement – The decision value of all correct processes is the same
  – Integrity – If the correct processes all proposed the same value, then any correct process in the decided state has chose that value
Consensus in synchronous systems

- Up to $f$ of the $N$ processes exhibit crash failures
- Algorithm proceeds in $f+1$ rounds, in each the correct processes B-multicast the values between themselves
  - Round duration is set based on max time for a correct process to multicast a message

Algorithm for process $p_i$

On initialization: $\text{Values}_{i}^1 := \{v_i\}$; $\text{Values}_{i}^0 := \{\}$

$\text{Values}^r_i$ holds the proposed values known to $p_i$ at the beginning of round $r$

1. In round $r$ from $[1,f+1]$
2. B-multicast($g$, $\text{Values}_{i}^r$ – $\text{Values}_{i}^{r-1}$) // send only new values
3. $\text{Values}_{i}^{r+1} := \text{Values}_{i}^r$
4. while (in round $r$) {
5. On B-deliver($V_j$) from some $p_j$
6. Insert $V_j$ in $\text{Values}_{i}^{r+1}$
7. }
8. After $(f+1)$ rounds
9. $d_i$ is minimum($\text{Values}_{i}^{f+1}$)
Consensus in synchronous systems

- Termination is obvious because the system is synchronous (fix number of rounds of fixed length)

- Agreement and integrity holds if all processes arrive at the same set of values at the end of the final round
  - By contradiction, if two processes differ in their final set, some correct process \( p_i \) has a value \( v \) that another correct process \( p_j \) does not
  - Only possible explanation, a process \( p_k \) managed to send \( v \) to \( p_i \), but didn’t get to \( p_j \), before crashing
  - But for that, \( p_k \) must have received it in a previous round in which \( p_j \) didn’t, so there must be a process \( p_l \) ...
  - And on and on, for each of the \( f+1 \) round,
  - But we have assumed at least \( f \) crashes!
Byzantine general problem

- Three or more generals have to agree to attack/retreat
  - One, the commander, issues the order to \( n-1 \) lieutenants
  - The lieutenants must decide whether to attack or retreat
  - One or more generals, \( m \), may be “treacherous” (faulty)

- If the commander is a traitor
  - Can propose attack to one lieutenant and retreat to another

- If the lieutenant is a traitor,
  - Can tell one lieutenant that the commander said attack and tell another that he said retreat

- A variant of consensus with a distinguished process proposing a value
  - Integrity – if the commander is correct, all correct processes agree with what he proposed
Byzantine generals problem – a solution

- The algorithm $\text{BG}(m)$ solves the BGP for $3m+1$ or more generals in the presence of at most $m$ traitors

Algorithm $\text{BG}(0)$
1. The commander (C) sends her value to every lieutenant
2. Lieutenants use the value received from C or RETREAT if they received no value

Algorithm $\text{BG}(m)$, $m>0$
1. C sends her value to every lieutenant
2. For each $i$, let $v_i$ be the value Lieutenant $i$ receives from C or RETREAT. Lieutenant $i$ acts as C in algorithm $\text{BG}(m-1)$ to send the value $v_i$ 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 algorithm $\text{BG}(m-1)$), or else RETREAT. Lieutenant $i$ uses the value $\text{majority}(v_1, \ldots, v_{n-1})$
A run with $N \geq 4$ and $f = 1$

- **A lieutenant is the traitor**
  - $L1$: $\text{majority}(v,v,x) = v$
  - $L2$: $\text{majority}(v,v,y) = v$

- **The commander is the traitor**
  - $L1$: $\text{majority}(v,w,z) = \text{NIL}$
  - $L2$: $\text{majority}(v,w,z) = \text{NIL}$
  - $L3$: $\text{majority}(v,w,z) = \text{NIL}$

- **What would happen if they sign their messages?**
Consensus, synchronous & asynchronous

- We discussed solutions for consensus and Byzantine generals problem
  - Our solutions assume synchronous system

- What if the system is asynchronous?
  - No algorithm can guarantee to reach consensus with even one faulty process [Fischer et. al, 1985]
  - Basic idea: how can you tell the process has crashed or things are just running slow?

- How do we work around this?
  - Masking faults: process keep data in persistent storage so that they can restart after crashing
  - Using perfect by design failure detectors – processes agree to a maximum response time (otherwise, the process has failed)
  - ...
Summary

- Agreement on either group communication or a common decision is key to many distributed systems.

- Designers can save time by knowing the situations when no solution is possible:
  - Synchronous systems + reliable message delivery + Byzantine failures – possible if $< 1/3$ of the processes fail.
  - Asynchronous systems + reliable message delivery + crash failures – not possible even if 1 process fails.