Distributed systems
Lecture 5: Consistent cuts, process groups, and mutual exclusion

Dr Robert N. M. Watson
• Saw physical time can’t be kept exactly in sync; instead use **logical clocks** to track ordering between events:
  – Defined $a \rightarrow b$ to mean ‘$a$ happens-before $b$’
  – Easy inside single process, & use causal ordering ($send \rightarrow receive$) to extend relation across processes
  – if $send_i(m_1) \rightarrow send_j(m_2)$ then $deliver_k(m_1) \rightarrow deliver_k(m_2)$

• **Lamport clocks**, $L(e)$: an integer
  – Increment to $\max$ of (sender, receiver)) + 1 on receipt
  – But given $L(a) < L(b)$, know nothing about order of $a$ and $b$

• **Vector clocks**: list of Lamport clocks, one per process
  – Element $V_i[j]$ captures #events at $P_j$ observed by $P_i$
  – Crucially: if $V_i(a) < V_j(b)$, can infer that $a \rightarrow b$ , and
    if $V_i(a) \sim V_j(b)$, can infer that $a \sim b$
Vector clocks: example

- When P2 receives \( m_1 \), it merges the entries from P1’s clock
  - choose the maximum value in each position
- Similarly when P3 receives \( m_2 \), it merges in P2’s clock
  - this incorporates the changes from P1 that P2 already saw
- Vector clocks **explicitly track the transitive causal order**: \( f \)’s timestamp captures the history of \( a, b, c \) & \( d \)
Consistent global state

• We have the notion of “a happens-before b” (a → b) or “a is concurrent with b” (a ~ b)

• What about ‘instantaneous’ system-wide state?
  – distributed debugging, GC, deadlock detection, ...

• Chandy/Lamport introduced **consistent cuts**:
  – draw a (possibly wiggly) line across all processes
  – this is a consistent cut if the set of events (on the lhs) is closed under the happens-before relationship
  – i.e. if the cut includes event x, then it also includes all events e which happened before x

• In practical terms, this means every *delivered* message included in the cut was also *sent* within the cut
Consistent cuts: example

- Vertical cuts are always consistent (due to the way we draw these diagrams), but some curves are ok too:
  - providing we don’t include any receive events without their corresponding send events
- Intuition is that a consistent cut could have occurred during execution (depending on scheduling etc),
Observing consistent cuts

- Chandy/Lamport Snapshot Algorithm (1985)
- Distributed algorithm to generate a snapshot of relevant system-wide state (e.g. all memory, locks held, ...)
- Flood a special **marker message** $M$ to all processes; causal order of flood defines the cut
- If $P_i$ receives $M$ from $P_j$ and it has yet to snapshot:
  - It pauses all communication, takes local snapshot & sets $C_{ij}$ to ${}$
  - Then sends $M$ to all other processes $P_k$ and starts recording $C_{ik} = \{ \text{set of all post local snapshot messages received from } P_k \}$
- If $P_i$ receives $M$ from some $P_k$ after taking snapshot
  - Stops recording $C_{ik}$, and saves alongside local snapshot
- Global snapshot comprises all local snapshots & $C_{ij}$
- Assumes reliable, in-order messages, & no failures

Fear not! This is not examinable.
Process groups

• It is useful to build distributed systems with **process groups**
  – Set of processes on some number of machines
  – Possible to **multicast** messages to all members
  – Allows fault-tolerant systems even if some processes fail

• Membership can be **fixed** or **dynamic**
  – if dynamic, have explicit join() and leave() primitives

• Groups can be **open** or **closed**:
  – Closed groups only allow messages from members

• Internally can be structured (e.g. coordinator and set of slaves), or symmetric (peer-to-peer)
  – Coordinator makes e.g. concurrent join/leave easier...
  – ... but may require extra work to **elect** coordinator

---

*When we use **multicast** in distributed systems, we mean something stronger than conventional network multicasting using datagrams – do not confuse them.*
Group communication: assumptions

• Assume we have ability to send a message to multiple (or all) members of a group
  – Don’t care if ‘true’ multicast (single packet sent, received by multiple recipients) or “netcast” (send set of messages, one to each recipient)
• Assume also that message delivery is reliable, and that messages arrive in bounded time
  – But may take different amounts of time to reach different recipients
• Assume (for now) that processes don’t crash
• What delivery orderings can we enforce?
With **FIFO ordering**, messages from a particular process $P_i$ must be received at all other processes $P_j$ in the order they were sent
- e.g. in the above, everyone must see $m_1$ before $m_3$
- (ordering of $m_2$ and $m_4$ is not constrained)

Seems easy but not trivial in case of delays / retransmissions
- e.g. what if message $m_1$ to $P2$ takes a loooong time?

Hence receivers may need to **buffer** messages to ensure order
Receiving versus delivering

• Group communication middleware provides extra features above ‘basic’ communication
  – e.g. providing reliability and/or ordering guarantees on top of IP multicast or netcast

• Assume that OS provides receive() primitive:
  – returns with a packet when one arrives on wire

• **Received** messages either delivered or held back:
  – **Delivered** means inserted into delivery queue
  – **Held back** means inserted into hold-back queue
  – held-back messages are delivered later as the result of the receipt of another message...
Implementing FIFO ordering

- Each process $P_i$ maintains a message sequence number (SeqNo) $S_i$
- Every message sent by $P_i$ includes $S_i$, incremented after each send
  - not including retransmissions!
- $P_j$ maintains $S_{ji}$: the SeqNo of the last delivered message from $P_i$
  - If receive message from $P_i$ with SeqNo $\not= (S_{ji}+1)$, hold back
  - When receive message with SeqNo $= (S_{ji}+1)$, deliver it ... and also deliver any consecutive messages in hold back queue ... and update $S_{ji}$

```c
receive(M from Pi) {
    s = SeqNo(M);
    if (s == (S_{ji}+1) ) {
        deliver(M);
        s = flush(hbq);
        S_{ji} = s;
    } else holdback(M);
}
```

![Diagram of FIFO ordering](image-url)
Stronger orderings

• Can also implement FIFO ordering by just using a reliable FIFO transport like TCP/IP
• But the general ‘receive versus deliver’ model also allows us to provide stronger orderings:
  – **Causal ordering**: if event $\text{multicast}(g, m_1) \rightarrow \text{multicast}(g, m_2)$, then all processes will see $m_1$ before $m_2$
  – **Total ordering**: if any processes delivers a message $m_1$ before $m_2$, then all processes will deliver $m_1$ before $m_2$
• Causal ordering implies FIFO ordering, since any two multicasts by the same process are related by $\rightarrow$
• Total ordering (as defined) does *not* imply FIFO (or causal) ordering, just says that all processes must agree
  – Often want **FIFO-total** ordering (combines the two)
Causal ordering

- Same example as previously, but now causal ordering means that
  (a) everyone must see \( m_1 \) before \( m_3 \) (as with FIFO), and
  (b) everyone must see \( m_1 \) before \( m_2 \) (due to happens-before)
- Is this ok?
  - No! \( m_1 \rightarrow m_2 \), but \( P2 \) sees \( m_2 \) before \( m_1 \)
  - To be correct, must hold back (delay) delivery of \( m_2 \) at \( P2 \)
  - But how do we know this?
Implementing causal ordering

• Turns out this is pretty easy!
  – Start with receive algorithm for FIFO multicast...
  – and replace sequence numbers with vector clocks

- Some care needed with dynamic groups

\[
\begin{align*}
(0,0,0) &\rightarrow (1,0,0) \\
&\rightarrow (1,1,0) \\
&\rightarrow (2,0,2) \\
&\rightarrow (1,0,2) \\
&\rightarrow (1,1,0) \\
&\rightarrow (1,0,1) \\
&\rightarrow (1,0,2) \\
\end{align*}
\]

Have \((0,0,0) \neq (1,0,2)\), so must hold back \(m_2\) until missing events seen

Once \(m_1\) received, can deliver \(m_1\) and then \(m_2\)
Total ordering

• Sometimes we want all processes to see exactly the same, FIFO, sequence of messages
  – particularly for state machine replication (see later)

• One way is to have a ‘can send’ token:
  – Token passed round-robin between processes
  – Only process with token can send (if he wants)

• Or use a dedicated sequencer process
  – Other processes ask for global sequence no. (GSN), and then send with this in packet
  – Use FIFO ordering algorithm, but on GSNs

• Can also build non-FIFO total-order multicast by having processes generate GSNs themselves and resolving ties
Ordering and asynchrony

• FIFO ordering allows quite a lot of asynchrony
  – E.g. any process can delay sending a message until it has a batch (to improve performance)
  – Or can just tolerate variable and/or long delays

• Causal ordering also allows some asynchrony
  – But must be careful queues don’t grow too large!

• Traditional total order multicast not so good:
  – Since every message delivery transitively depends on every other one, delays holds up the entire system
  – Instead tend to an (almost) synchronous model, but this performs poorly, particularly over the wide area ;-
  – Some clever work on virtual synchrony (for the interested)
Distributed mutual exclusion

- In first part of course, saw need to coordinate concurrent processes / threads
  - In particular considered how to ensure mutual exclusion: allow only 1 thread in a critical section
- A variety of schemes possible:
  - test-and-set locks; semaphores; monitors; active objects
- But most of these ultimately rely on hardware support (atomic operations, or disabling interrupts...)
  - not available across an entire distributed system
- Assuming we have some shared distributed resources, how can we provide mutual exclusion in this case?
Solution #1: central lock server

• Nominate one process C as coordinator
  – If Pᵢ wants to enter critical section, simply sends lock message to C, and waits for a reply
  – If resource free, C replies to Pᵢ with a grant message; otherwise C adds Pᵢ to a wait queue
  – When finished, Pᵢ sends unlock message to C
  – C sends grant message to first process in wait queue
Central lock server: pros and cons

• Central lock server has some good properties:
  – **Simple** to understand and verify
  – **Live** (providing delays are bounded, and no failure)
  – **Fair** (if queue is fair, e.g. FIFO), and easily supports priorities if we want them
  – **Decent performance**: lock acquire takes one round-trip, and release is ‘free’ with asynchronous messages

• But C can become a performance bottleneck...

• ... and can’t distinguish crash of C from long wait
  – can add additional messages, at some cost
Solution #2: token passing

- Avoid central bottleneck
- Arrange processes in a logical ring
  - Each process knows its predecessor & successor
  - Single token passes continuously around ring
  - Can only enter critical section when possess token; pass token on when finished (or if don’t need to enter CS)
Token passing: pros and cons

• Several advantages:
  – Simple to understand: only 1 process ever has token => mutual exclusion guaranteed by construction
  – No central server bottleneck
  – Liveness guaranteed (in the absence of failure)
  – So-so performance (between 0 and N messages until a waiting process enters, 1 message to leave)

• But:
  – Doesn’t guarantee fairness (FIFO order)
  – If a process crashes must repair ring (route around)
  – And worse: may need to regenerate token – tricky!

• And constant network traffic: an advantage???
Solution #3: totally ordered multicast

• Scheme due to Ricart & Agrawala (1981)
• Consider $N$ processes, where each process maintains local variable state which is one of \{\texttt{FREE}, \texttt{WANT}, \texttt{HELD}\}
• To obtain lock, a process $P_i$ sets state := \texttt{WANT}, and then multicasts lock request to all other processes
• When a process $P_j$ receives a request from $P_i$:
  – If $P_j$’s local state is \texttt{FREE}, then $P_j$ replies immediately with \texttt{OK}
  – If $P_j$’s local state is \texttt{HELD}, $P_j$ queues the request to reply later
• A requesting process $P_i$ waits for \texttt{OK} from $N$-1 processes
  – Once received, sets state := \texttt{HELD}, and enters critical section
  – Once done, sets state := \texttt{FREE}, & replies to any queued requests
• What about concurrent requests?

By concurrent we mean: $P_j$ is already in the WANT state when it receives a request from $P_i$
Handling concurrent requests

• Need to decide upon a total order:
  – Each process maintains a Lamport timestamp, $T_i$
  – Processes put current $T_i$ into request message
  – Insufficient on its own (recall that Lamport timestamps can be identical) => use process id (or similar) to break ties

• Hence if a process $P_j$ receives a request from $P_i$ and $P_j$ has an outstanding request (i.e. $P_j$’s local state is £WANT£)
  – If $(T_j, P_j) < (T_i, P_i)$ then queue request from $P_i$
  – Otherwise, reply with £OK£, and continue waiting

• Note that using the total order ensures correctness, but not fairness (i.e. no FIFO ordering)
  – Q: can we fix this by using vector clocks?
 Totally ordered multicast: example

- Imagine $P_1$ and $P_2$ simultaneously try to acquire lock...
  - Both set state to $\text{WANT}$, and both send multicast message
  - Assume that timestamps are 17 (for $P_1$) and 9 (for $P_2$)
- $P_3$ has no interest (state is $\text{FREE}$), so replies Ok to both
- Since $9 < 17$, $P_1$ replies Ok; $P_2$ stays quiet & queues $P_1$’s request
- $P_2$ enters the critical section and executes...
- ... and when done, replies to $P_1$ (who can now enter critical section)
Additional details

• Completely unstructured decentralized solution ... but:
  – Lots of messages (1 multicast + N-1 unicast)
  – Ok for most recent holder to re-enter CS without any messages

• Variant scheme (Lamport) - **multicast for total ordering**
  – To enter, process P_i multicasts request(P_i, T_i) [same as before]
  – On receipt of a message, P_j replies with an ack(P_j, T_j)
  – Processes keep all requests and acks in ordered queue
  – If process P_i sees his request is earliest, can enter CS ... and
    when done, multicasts a release(P_i, T_i) message
  – When P_j receives release, removes P_i’s request from queue
  – If P_j’s request is now earliest in queue, can enter CS...

• Both Ricart & Agrawala and Lamport’s scheme have N points of failure: doomed if any process dies :-(

Summary + next time

• (More) vector clocks
• Consistent global state + consistent cuts
• Process groups and reliable multicast
• Implementing order
• Distributed mutual exclusion

• Leader elections and distributed consensus
• Distributed transactions and commit protocols
• Replication and consistency