Distributed Systems
8L for Part IB

Lecture 6

Dr Robert N. M. Watson

1

Last time

- Saw how we can build ordered multicast
  - Messages between processes in a group
  - Need to distinguish receipt and delivery
  - Can provide FIFO, Causal or Total (several variants)
- Considered distributed mutual exclusion:
  - Want to arrange only one process can enter CS
  - Central server ok; but potential bottleneck & SPoF
  - Token passing ok: but traffic, repair, token loss
  - Totally-Ordered Multicast …

2

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 { FREE, WANT, HELD }
- To obtain lock, a process P_i sets state:= 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 FREE, then P_j replies immediately with **OK**
  - If P_j's local state is HELD, P_j queues the request to reply later
- A requesting process P_i waits for **OK** from N-1 processes
  - Once received, sets state:= HELD, and enters critical section
  - Once done, sets state:= FREE, & replies to any queued requests
- What about **concurrent requests**?

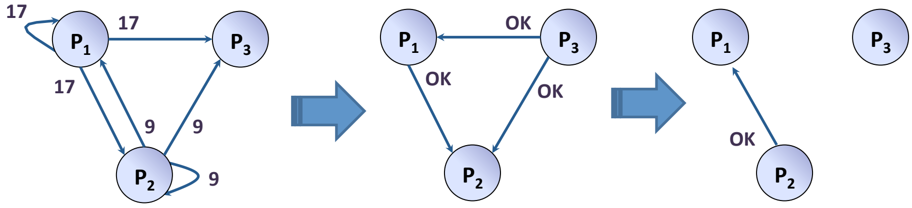
3

Handling Concurrent Requests

- Need to decide upon a total order:
  - Each processes 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?

4

Totally-Ordered Multicast: Example



- Imagine P1 and P2 simultaneously try to acquire lock…
  - Both set state to WANT, and both send multicast message
  - Assume that timestamps are 17 (for P1) and 9 (for P2)
- P3 has no interest (state is FREE), so replies Ok to both
- Since 9 < 17, P1 replies Ok; P2 stays quiet & queues P1's request
- P2 enters the critical section and executes…
- … and when done, replies to P1 (who can now enter critical section)

5

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 (due to Lamport):
  - 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…
- Note that both Ricart & Agrawala and Lamport's scheme, have N points of failure: doomed if *any* process dies :-(

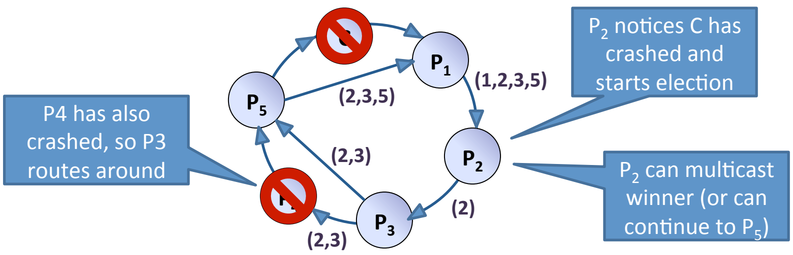
6

Leader Election

- Many schemes are built on the notion of having a well-defined 'leader' (master, coordinator)
  - examples seen so far include the Berkeley time synchronization protocol, and the central lock server
- An election algorithm is a dynamic scheme to choose a unique process to play a certain role
  - assume P_i contains state variable **elected$_i$**
  - when a process first joins the group, **elected$_i$** = UNDEFINED
- By the end of the election, for every Pi,
  - **elected$_i$** = P_x, where P_x is the winner of the election, or
  - **elected$_i$** = UNDEFINED, or
  - P_i has crashed or otherwise left the system

7

Ring-Based Election



- System has coordinator who crashes
- Some process notices, and starts an election
  - Puts its id into a message, and sends to its successor
  - On receipt, a process acks to sender (not shown), and then appends its id and forwards the election message
  - Finished when a process receives message containing its id

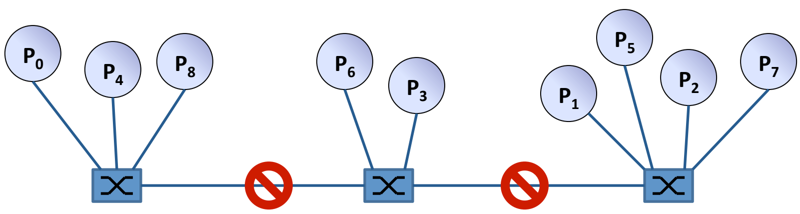
8

The Bully Algorithm

- Assume that we know the ids of all processes
- Algorithm proceeds by attempting to elect the process still alive with the highest id
  - Assumes we can reliably detect failures by timeouts
- If process P_i sees current leader has crashed, sends **election** message to all processes with higher ids, and starts a timer
  - Concurrent election initiation by multiple processes is fine
  - Processes receiving an election message reply **OK** to sender, and start an election of their own (if not already in progress)
  - If a process hears nothing back before timeout, it declares itself the winner, and multicasts result
- A dead process that recovers (or new process that joins) also starts an election: can ensure highest ID always elected

9

Problems with Elections



- Algorithms rely on being able use timeouts to reliably detect failure
- However it is possible for networks to fail: a **network partition**
  - Some processes can speak to others, but not all
- Can lead to **split-brain syndrome**:
  - Every partition independently elects a leader => too many bosses!
- To fix, need some secondary (& tertiary?) communication scheme
  - e.g. secondary network, shared disk, serial cables, …

10

Aside: Consensus

- Elections are a specific example of a more general problem: **consensus**
  - Given a set of N processes in a distributed system, how can we get them all to agree on something?
- Classical treatment has every process P_i propose something (a value V_i)
  - Want to arrive at some deterministic function of V_i's (e.g. 'majority' or 'maximum' will work for election)
- A correct solution to consensus must satisfy:
  - **Agreement**: all nodes arrive at the same answer
  - **Validity**: answer is one that was proposed by someone
  - **Termination**: all nodes eventually decide

11

"Consensus is impossible"

- Famous result due to Fischer, Lynch & Patterson (1985)
  - Focuses on an asynchronous network (unbounded delays) with at least one process failure
  - Shows that it is possible to get an infinite sequence of states, and hence never terminate
  - Given the Internet is an asynchronous network, then this seems to have major consequences!!
- Not really:
  - Result actually says we can't always guarantee consensus, not that we can never achieve consensus
  - And in practice, we can use tricks to mask failures (such as reboot, or replication), and to ignore asynchrony
  - Have seen solutions already, and will see more later

12

Transaction Processing Systems

- Last term looked at **transactions**:
  - Support for composite operations (i.e. a collection of reads and updates to a set of objects)
- A transaction is **atomic** ("all-or-nothing")
  - If it commits, all operations are applied
  - If it aborts, it's as if nothing ever happened
- A committed transaction moves system from one **consistent** state to another
- Transaction processing systems also provide:
  - **isolation** (between concurrent transactions)
  - **durability** (committed transactions survive a crash)

13

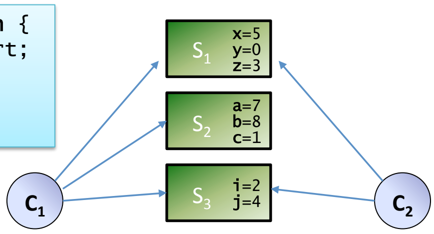
Distributed Transactions

- Scheme described last term was client/server
  - (even though I didn't say it at the time ;-)
  - Clients communicate with a server (e.g. a database)
- However ***distributed transactions*** are those which span *multiple* transaction processing servers
- E.g. booking a complicated trip from London to Vail, CO
  - Could fly LHR -> LAX -> EGE + hire a car…
  - … or fly LHR -> ORD -> DEN + take a public bus
- Want a complete trip (i.e. atomicity)
  - Not get stuck in an airport with no onward transport!
- Must coordinate actions across multiple parties

14

A Model of Distributed Transactions

```
T1 transaction {
  if(x<2) abort;
  a:= z;
  j:= j + 1;
}
```




```
T2 transaction {
  z:= (i+j);
}
```

- Multiple servers (S_1, S_2, S_3, …), each holding some objects which can be read and written within client transactions
- Multiple concurrent clients (C_1, C_2, …) who perform transactions which interact with one or more servers
  - e.g. T1 reads **x**, **z** from S_1, writes **a** on S_2, and reads & writes **j** on S_3
  - e.g. T2 reads **i**, **j** from S_3, then writes **z** on S_1
- A successful commit implies agreement at all servers

15

Implementing Distributed Transactions

- Can build on top of solution for single server:
  - e.g. use locking or shadowing to provide isolation
  - e.g. use write-ahead long for durability
- Main additional challenge is in coordinating decision to either commit or abort
  - Assume clients create unique transaction id: **TxID**
  - Uses TxID in every read or write request to a server S_i
  - First time S_i sees a given TxID, it starts a tentative transaction associated with that transaction id
  - When client wants to commit, must perform **atomic commit** of all tentative transactions across all servers

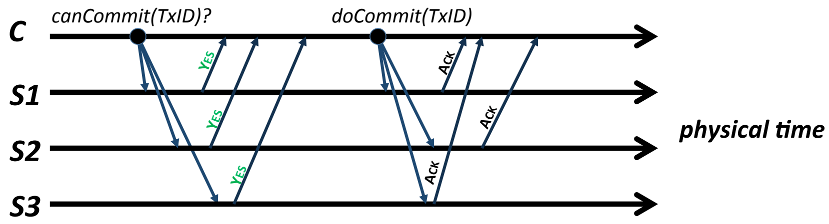
16

Atomic Commit Protocols

- A naïve solution would have client simply invoke **commit**(TxID) on each server in turn
  - Will work only if no concurrent conflicting clients, every server commits (or aborts), and no server crashes
- To handle concurrent clients, introduce a **coordinator**:
  - A designated machine (can be one of the servers)
  - Clients ask coordinator to commit on their behalf… and hence coordinator can **serialize** concurrent commits
- To handle inconsistency/crashes, coordinator:
  - Asks all involved servers if they *could* commit TxID
  - Servers S_i reply with a vote V_i = { COMMIT, ABORT }
  - If all V_i = COMMIT, coordinator multicasts **doCommit**(TxID)
  - Otherwise, coordinator multicasts **doAbort**(TxID)

17

Two-Phase Commit (2PC)



- This scheme is called **two-phase commit (2PC)**:
  - First phase is **voting**: collect votes from all parties
  - Second phase is **completion**: either abort or commit
- Doesn't require ordered multicast, but needs reliability
  - If server fails to respond by timeout, treat as a vote to abort
- Once all ACKS received, inform client of successful commit

18

2PC: Additional Details

- Client (or any server) can abort during execution: simply multicasts **doAbort**(TxID) to all servers
- If a server votes to abort, can immediately abort locally
- If a server votes to commit, it must be able to do so if subsequently asked by coordinator:
  - Before voting to commit, server will **prepare** by writing entries into log and flushing to disk
  - (this is why some sources call the first phase “prepare”)
  - Also records all requests from & responses to coordinator
  - Hence even if crashes *after* voting to commit, will be able to recover on reboot

19

2PC: Coordinator Crashes

- Coordinator must also persistently log events:
  - Including initial message from client, requesting votes, receiving replies, and final decision made
  - Lets it reply if (rebooted) client or server asks for outcome
  - Also lets coordinator recover from reboot, e.g. re-send any vote requests without responses, or reply to client
- One additional problem occurs if coordinator crashes after phase 1, but before initiating phase 2:
  - servers will be uncertain of outcome…
  - if voted to commit, will have to continue to hold locks, etc
- (other consensus protocols such as 3PC provide better progress guarantees if permanent failure can happen)

20

Replication

- Many distributed systems involve replication
  - Multiple copies of some object stored at different servers
  - Multiple servers capable of providing some operation(s)
- Three key advantages:
  - **Load-Balancing**: if have many replicas, then can spread out work from clients between them
  - **Lower Latency**: if replicate an object/server close to a client, will get better performance
  - **Fault-Tolerance**: can tolerate the failure of some replicas and still provide service
- Examples include DNS, web & file caching (& content-distribution networks), replicated databases, …

21

Replication in a Single System

- One good example is RAID:
  - RAID = Redundant Array of Inexpensive Disks
  - i.e. disks are cheap, so use several instead of just one
  - if replicate data across disks, can tolerate disk crash
- A variety of different configurations (levels)
  - RAID 0: **stripe** data across disks, i.e. block 0 to disk 0, block 1 to disk 1, block 2 to disk 0, and so on
  - RAID 1: **mirror** (replicate) data across disks, i.e. block 0 written on disk 0 and disk 1
  - RAID 5: **parity** – write block 0 to disk 0, block 1 to disk 1, and (block 0 XOR block 1) to disk 2
- Get improved performance since can access disks in parallel
- With RAID 1, 5 also get fault-tolerance