CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Replication

- A *fundamental concept* with many uses
  - If we can solve this core problem, we can apply the solution in many settings
- Replication is a basic primitive... But one missing in most development toolkits
  - We find replication mechanisms *inside* the operating system (e.g. IBM WebSphere uses replication, as does Microsoft's Windows Clustering technology...)
  - End-users often given weaker solutions (pub-sub)

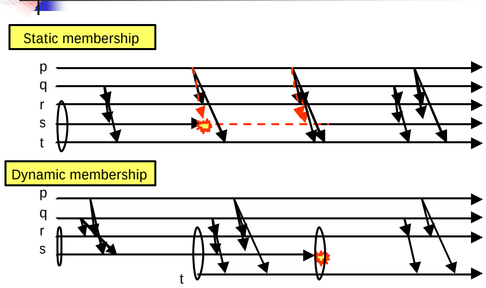
Uses of replication

- Replicate data or a service for high availability
- Replicate data so that group members can share loads and improve scalability
- Replicate locking or synchronization state
- Replicate membership information in a data center so that we can route requests
- Replicate management information or parameters to tune performance

Who "does" the replication?

- We think of replication as happening inside *groups*
  - Could be a group of identical components
  - Or just a group of processes that asked to join in order to replicate a data structure
    - Members might be different programs...
- Sometimes we know who might be a replica ahead of time ("static model"), sometimes not ("dynamic model")

Two replication models



Issues raised?

Static model

- Often only a subset of the "replicas" are running
- Need to agree on order of concurrent updates
- Must deal with inconsistency in replicas that were down for a while but now have recovered

Dynamic model

- Membership changes as members join/fail/leave
  - "Report" view changes
  - How to detect failure?
  - How to initialize replica (joining member)?
- Need to agree on order of concurrent updates
- Normally, failed members replaced by other processes

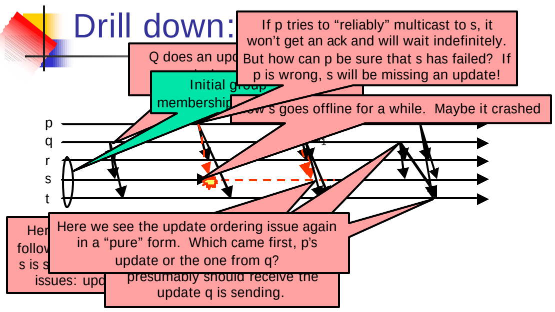
Further issues raised

- Does choice of model
  - Impact performance?
  - Impact platform complexity?
  - Impact system-wide design philosophy, e.g. degree of use of end-to-end ideas?

Let's focus on update ordering

- We want to
  - Replicate data
  - Update it while accessing it
- What sorts of issues must be addressed?

Drill down:



Questions to ask about order

- Who should receive an update?
- What update ordering to use?
- How expensive is the ordering property?

Questions to ask about order

- Delivery order for concurrent updates
  - Issue is more subtle than it looks!
  - We can fix a system-wide order, but...
    - Sometimes nobody notices out of order delivery
    - System-wide ordering is expensive
    - If we care about speed we may need to look closely at cost of ordering

Ordering example

- System replicates variables x, y
  - Process p sends "x = x/2"
  - Process q sends "x = 83"
  - Process r sends "y = 17"
  - Process s sends "z = x/y"
- To what degree is ordering needed?

Ordering example

- x=x/2   x=83
- These clearly "conflict"
  - If we execute x=x/2 first, then x=83, x will have value 83.
  - In opposite order, x is left equal to 41.5

Ordering example

- x=x/2   y=17
- These don't seem to conflict
  - After the fact, nobody can tell what order they were performed in

Ordering example

- z=x/y
- This conflicts with updates to x, updates to y and with other updates to z

Commutativity

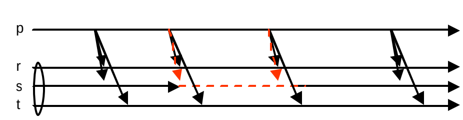
- We say that operations "commute" if the final effect on some system is the same even if the order of those operations is swapped
- In general, a system worried about ordering concurrent events need not worry if the events commute

Single updater

- In many systems, there is only one process that can update a given type of data
  - For example, the variable might be "sensor values" for a temperature sensor
  - Only the process monitoring the sensor does updates, although perhaps many processes want to read the data and we replicate it to exploit parallelism
  - Here the only "ordering" that matters is the FIFO ordering of the updates emitted by that process

Single updater

- If p is the only update source, the need is a bit like the TCP "fifo" ordering



Mutual exclusion

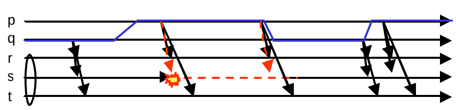
- Another important case we'll study closely
- Arises in systems that use locks to control access to shared data
  - This is very common, for example in "transactional" systems (we'll discuss them next week)
  - Very often without locks, a system rapidly becomes corrupted

Mutual exclusion

- Suppose that before performing conflicting operations, processes must lock the variables
- This means that there will never be any true concurrency
- And it simplifies our ordering requirement

Mutual exclusion

- Dark blue when holding the lock



- How is this case similar to "FIFO" with one sender? How does it differ?

Mutual exclusion

- Are these updates in "FIFO" order?
  - No, the sender isn't always the same
  - But yes in the sense that there is a unique path through the system (corresponding to the lock) and the updates are ordered along that path
- Here updates are ordered by Lamport's happened before relation: $\rightarrow$

Types of ordering we've seen

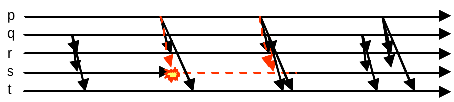
- *cheapest* — Deliver updates in an order matching the FIFO order in which they were sent
- *Still cheap* — Deliver updates in an order matching the $\rightarrow$ order in which they were sent
- *More costly* — For conflicting concurrent updates, pick an order and use that order at all replicas
- *Most costly* — Deliver an update to all members of a group according to "membership view" determined by ordering updates wrt view changes

Types of ordering we've seen

- ***fbcast*** — Deliver updates in an order matching the FIFO order in which they were sent
- ***cbcast*** — Deliver updates in an order matching the $\rightarrow$ order in which they were sent
- ***abcast*** — For conflicting concurrent updates, pick an order and use that order at all replicas
- ***gbcast*** — Deliver an update to all members of a group according to "membership view" determined by ordering updates wrt view changes

Now continue to "drill down"

- We drilled down on ordering
- But what about failure?



What makes it hard?

- Detecting a failure is very tricky
  - Network can lose messages...
  - ... a machine can be briefly disconnected from the network
  - ... or could experience a brief overload causing it to run slow, or ignore incoming messages, or "freeze up"
- Are these transient problems "failures"?

Transient versus real failures

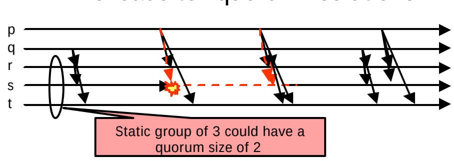
- A real failure persists long enough so that the system has no choice except to move on
  - It may take over some roles from failed process (a backup could become primary)
  - Although the primary might recover, it won't be the primary server anymore!
- A transient failure repairs itself before irrevocable compensating events occur

Are there any "real" failures?

- This leads to a view that failure isn't absolute
  - In fact we can't distinguish a failed machine from one that is simply not responding
  - And this is fundamental
- Are there options other than true failure detection?

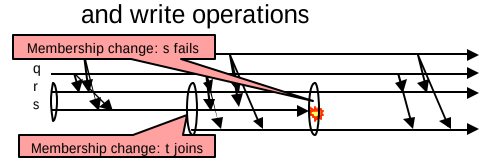
Options for coping with faults

- One idea is to build a system so that if a majority of processes is running, the system can continue to run
- This leads to "quorum" solutions



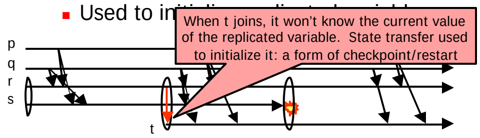
Options for coping with faults

- Another approach lets the group "manage" its membership dynamically
- We do need a quorum vote when membership changes... but not for read and write operations



Options for coping with faults

- We also need to deal with processes that join (or recover and rejoin) a group
- This involves not just a membership change but also a "state transfer"
  - Used to in



Types of ordering we've seen

- Order of an update relative to a membership change
  - When process s crashed, p no longer needed to wait for it to acknowledge updates
  - When s recovers, p must again send it updates. And it needs to fix any data that was updated while it was down

Peek ahead in CS514

- We'll build solutions to these problems
- One way: quorum protocols + 2PC
  - Group members are a subset of some list
  - Read and update quorums must overlap. For example, read operation "visits" 2 members; updates "visits" N-1 members
  - 2PC needed as part of the update protocol
- Second way: use quorum methods only to manage "group view" membership
  - Enables high-speed multicast protocols and cheaper reads

Recommended readings

- In the textbook, we're at the beginning of Part III (Chapter 14)
- But transactional model is covered in Chapters 6 and 22
- For next week will focus on that material