

Lecture 7 – Process Groups, Causal Ordering

1. Questions from reviews
2. Overall model
 - a. Small scale distributed system: air traffic control
 - i. Radars sense where planes are, send out updates
 - ii. Controllers make requests, send out their commands
 - iii. Planes ask for commands
 - iv. Note that radars + planes are “outside” – the system is really the controllers
 - b. QUESTION: What are goals?
 - i. Goal is fault tolerant computing
 1. Use replication for reliability
 - ii. Goal is simple programming
 1. Programmer relies on library/service to handle things
 - iii. Non goal: byzantine fault tolerance
 1. Rely on failure detector to mark failed nodes as dead
 - 2. Most code doesn't worry about failures/timeouts.**
 - c. USE:
 - i. Used in DCE corba for dist object-oriented systems
 - ii. Used in Microsoft cluster service for coordination
 - iii. Used by stock exchange, French air-traffic control
 - iv. Ultimately lost in the market to much larger scheme for database-oriented solutions
3. History of model
 - a. Grew out of the idea of replicated state machines, atomic delivery of messages
 - b. Want to adapt to a practical setting – not just replicated, deterministic state machine, but any applications
 - c. Want to make higher performance than atomic/total ordering
4. WHAT does the model include?
 - a. Failure mode: halt (fail stop)
 - i. Processes fail by halting
 - ii. A failure detector service detects failures, sends out notification messages
 - b. Process groups
 - i. Names for groups (e.g. identifiers)
 - ii. Memberships change over time
 1. Unlike byzantine generals...
 - c. Reliable Multicast (called broadcast) to a group
 - i. Can achieve “atomic broadcast” meaning all receive or none do
 1. Just like byzantine generals

- ii. Relaxed a bit: if one node receives a message then fails before sending anything else, order can be changed at other nodes
- d. Ordering model: virtual synchrony
 - i. Ideal situation: clocks advance in lockstep on all nodes. Reality is clock skew, message delay, processing delays
 - 1. Everything has a total global order
 - ii. Implement *virtual synchrony*, which has the same programming model as synchrony

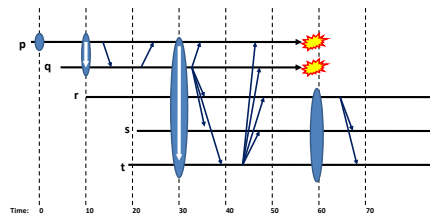


Fig. 6.1 Synchronous run.

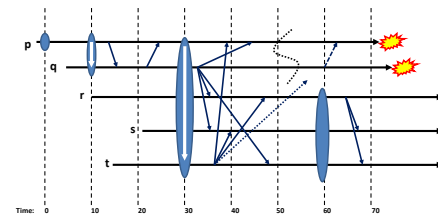


Fig. 6.2 Virtually synchronous run.

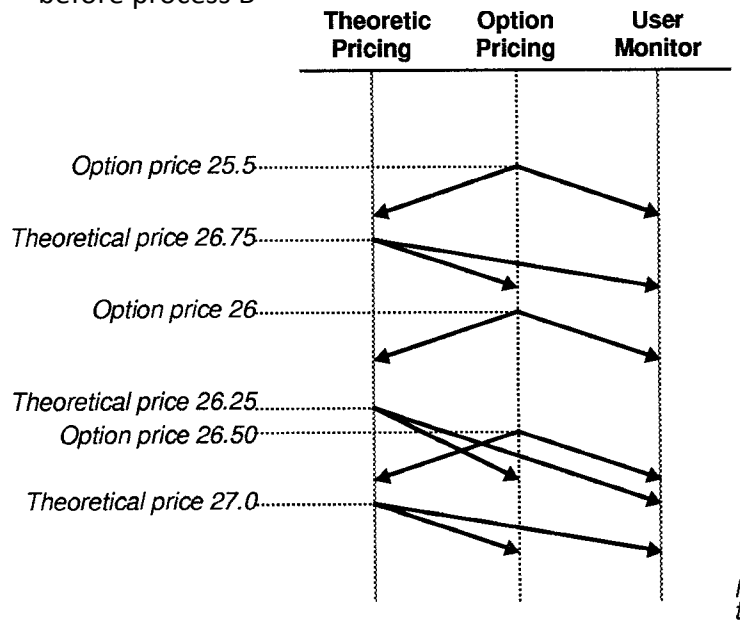
- iii.
 - 1. Difference: concurrent messages can overlap

5. Process groups

- a. QUESTION: what is the point?
 - i. Naming: keep track of who is interested in an object
 - ii. Membership: handle views of who is supposed to receive messages
 - iii. Failure reporting: other members of a group learn of failed members
- b. USES:
 - i. Diffusion groups: propagate information from leader to followers
 - ii. Client server groups: clients talk to a group of servers
 - iii.
- c. QUESTION: How are process groups maintained?
 - i. GBCAST (Group broadcast) communicates membership changes
 - ii. QUESTION: How should these be ordered with respect to normal application communication?
 - 1. A: want total order (like a distributed snapshot): app messages are either before membership change or afterwards
 - iii. Basic model: failure detector service runs at every node
 - 1. When app detects possible failure (e.g. missed message), notifies failure detector
 - 2. Failure detector can then use GBCAST to make failure visible to all
- d. Protocol for updating view:
 - i. Send "view extension message"
 - 1. On receipt, if no prior concurrent view extension, than ACK
 - 2. Else NACK, providing nodes from other view extension
 - ii. On receipt of ACKs
 - 1. Send out commit making new view real
 - iii. On receipt of a NACK, update extension and retry from beginning
 - iv. If there are partial views from a failed extension

1. If new primary has them, include failure of prior manager, includes in view (to prevent NACK)
 2. If has committed prior extension, some nodes may not have committed – includes in next view.
- e. QUESTION: How use process groups
- i. Keep track of coordination information (e.g. GFS masters in Google File System)
 - ii. Different terminals used by different air traffic controllers
6. Multicast Primitives
- a. Key idea: virtual synchrony
- i. In real synchrony, can only send one message at a time (to get total order everywhere)
 - ii. In virtual synchrony, can have concurrent independent operation, but ensure delivery is in correct order at the end
 1. Buffer messages at recipient until can be delivered in right order
 - iii. SO: separate reception (message arrives) from delivery (give to application)
 - iv.
- b. GBCAST: totally ordered with respect to other communication
- i. Messages from a failed process must be **delivered** before GBCAST of its failure
 - ii. GBCASTS and other broadcasts with overlapping destinations must have same order
 - iii. NOTE: this ordering requirement (ordered with everything) could be very expensive!
 - iv. IMPLEMENTATION: deferred
- c. ABCAST: atomic broadcast
- i. Specify a destination label (scope of ordering) so you can have independent atomic broadcasts going on
 1. Want most flexibility possible in ordering
 - ii. All ABCAST delivered to all destinations or none (Atomic)
 1. If delivered to one node & sender fails, receiver can resend
 - iii. All ABCAST to same label are received in same order at all destinations
 - iv. Prototype implementation: two phase delivery (like Lamport)
 1. Send msg to all recipients
 2. Recipients mark **undelivered**, send back a priority (e.g. like a lamport clock)
 3. Sender collects all acks, picks max priority and sends it back
 4. Receiver resorts queue, marks message **deliverable** and delivers message at head of queue
 5. NOTE: single queue undelivered and deliverable messages
 6. SHOW EXAMPLE
 7. NOTE: can have a separately delivery queue for each label
 - v. Reliability:

1. If a node has an undelivered message and detects failure of sender, will resend as the new leader (guarantees eventual delivery if any recipient received it).
- d. CBCAST: causal broadcast
 - i. Specify set of destinations. (process group)
 - ii. Ordering:
 1. Ensures happens-before delivery: if message sent by A to B and C, then B sends a message to C, then C receives message from A before message from B
 2. Uses “clabel” to express causality, like Vector or Lamport clocks
 3. QUESTION: Why?
 - a. Suppose you have a file
 - i. Process A multicasts “create file F”
 - ii. Process B multicasts “append to file F”
 - b. Causality ensures that all members get process A message before process B

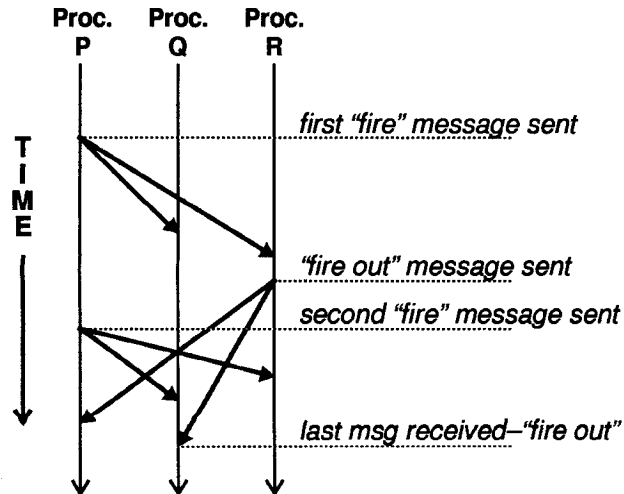


- 4.
5. Notice: does not ensure total order (P1 sees broadcast in 4 and 5 an order different from P2 and P3)
6. Example: doesn't provide total order,
7. VISION: FIFO channels in point-to-point are helpful (e.g. tcp/ip)
 - a. Ensure things come in the right order
 - i. Buffer things that arrive out of order, resend if missed
 - b. Want same property for multicast, but want most useful relaxed order (for performance)
- iii. Atomic delivery: to all or none of destination
- e. Use of broadcast:

- i. ABCAST,GBCAST: tend to be synchronous to do things like to do an RPC that updates common state
 - 1. Use it for performing totally ordered writes
- ii. CBCAST: tends to be async: fire & forget
 - 1. E.g. read an object by “registering” a read lock with CBCAST and reading a local copy
 - 2. Can then read local copy & drop lock
 - 3. Is totally ordered before or after other ABCASTS
 - 4. Can use for a lock:
 - a. Broadcast to acquire lock, holder replies to oldest broadcast
 - b. Causality ensures lock arrives after any messages preceding lock release
 - c. Same idea as Lamport lock, but use causal broadcast instead of atomic

7. Objections

- a. David Cheriton and Dale Skeen had a paper in SOSP’1993 saying causally & totally ordered communication is not very helpful:
 - i. Fundamental problem: causality is around communications, but doesn’t respect real ordering of program (e.g. database serializability), doesn’t handle stable updates to persistent data
 - 1. Their view: you have durable data and separate processes operating it (like a database)
 - 2. Want consistent updates to stored data
 - 3. CATOCS doesn’t really do this.
 - ii. Does not recognize causality outside the system (e.g. between sensors/actuators in real world.)
 - 1. Example: fire detected (broadcast), fire out (broadcast in response). Second fire detected (broadcast) could overlap – does not preserve causality when events are causally ordered externally



- 2.
3. Problem: causality of second fire starting after first not respected
- iii. Cannot group updates like a transaction
 1. Suppose updating multiple objects – need to acquire a lock (like lamport clock paper)
- iv. Cannot expose semantic orderings outside of messages
 1. E.g. stock pricing: exposes causal order, but if that isn't the right order (e.g. A sends to B and C, B sends to C, A sends to C after B in stock pricing), then not enough
- v. Inefficient
 1. May need to buffer messages before delivery (e.g. ABCAST, CBCAST)
- b. Responses from Birman
 - i. Focus on apps without durable state – they work well with a database—and more on command/control with short-term transient state
 1. E.g. who is the leader now, who is holding locks right now
 2. Tend not to have multi-object updates as in a database
 3. Database apps interact indirectly through shared objects
 - a. E.g. write/read file in file system, update/query data
 4. Control apps interact directly
 - a. Send message to processes telling them what to do.
 - ii. Most causality actually captured by communication
 - iii. Can do transactions with a CBCAST locks: get lock, then CBCAST updates asynchronously
 - iv. Inefficient: can condense down to a vector clock per message, not very big.
 1. Any kind of ordered delivery requires some buffering plus clocks
 2. E.g. windows for TCP/IP
 3. Question: can cost be small, can benefit outweigh cost?