Distributed Coordination

Topics

- Event Ordering
- Mutual Exclusion
- Atomicity of Transactions– Two Phase Commit (2PC)
- Deadlocks
  - Avoidance/Prevention
  - Detection
- The King has died. Long live the King!

Event Ordering

Coordination of requests (especially in a fair way) requires events (requests) to be ordered.

- Stand-alone systems:
  - Shared Clock / Memory
  - Use a time-stamp to determine ordering
- Distributed Systems
  - No global clock
  - Each clock runs at different speeds

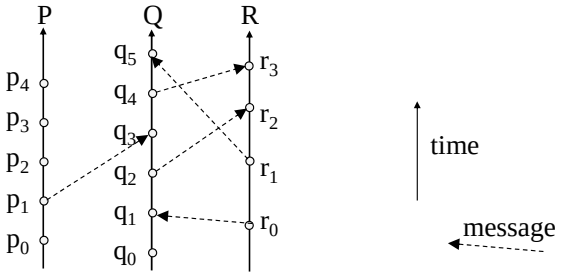
How do we order events running on physically separated systems?

- Messages (the only mechanism for communicating between systems) can only be received after they have been sent.

Event Ordering: Happened Before Relation

1. If A and B are events in the same process, and A executed before B, then $A \rightarrow B$.
2. If A is a message sent and B is when the message is received, then $A \rightarrow B$.
3. $A \rightarrow B$, and $B \rightarrow C$, then $A \rightarrow C$

Happened-Before Relationship



- Ordered events
  - p_1 preceeds ___
  - q_4 preceeds ___
  - q_2 preceeds ___
  - p_0 preceeds ___
- Unordered (Concurrent) events
  - q_0 is concurrent with ___
  - q_2 is concurrent with ___
  - q_4 is concurrent with ___
  - q_5 is concurrent with ___

Happened Before and Total Event Ordering

Define a notion of event ordering such that:

1. If A $\rightarrow$ B, then A precedes B.
2. If A and B are concurrent events, then nothing can be said about the ordering of A and B.

- Solution:

1. Each processor i maintains a logical clock LC_i
2. When an event occurs locally, ++ LC_i
3. When processor X sends a message to Y, it also sends LC_x in the message.
4. When Y receives this message, it:

```
if LCy < (LCx + 1) LCy  = LCx + 1;
```

Note: If “time” of A precedes “time” of B, then ???

If A->B and C->B does A->C?

1. Yes
2. No

Mutual Exclusion – Centralized Approach

One known process in the system coordinates mutual exclusion

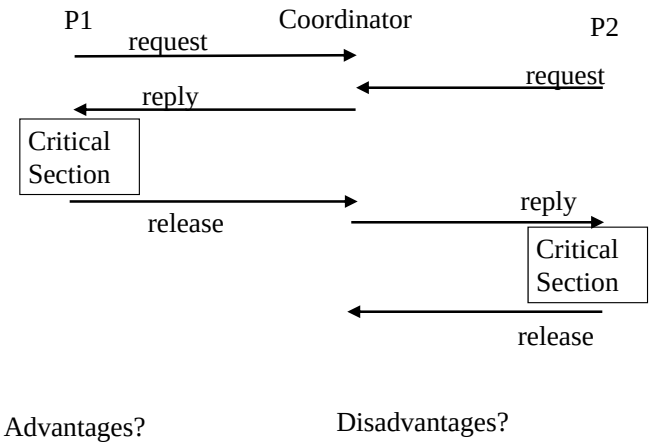
Client:

- Send a request to the controller, wait for reply
- When reply comes, back – enter critical section
- When finished, send release to controller.

Controller:

- Receives a request: If mutex is available, immediately send a reply (and mark mutex busy with client id). Otherwise, queue request.
- Receives a release from current user: Remove next requestor from queue and send reply. Otherwise, mark mutex available.

Example: Centralized Approach



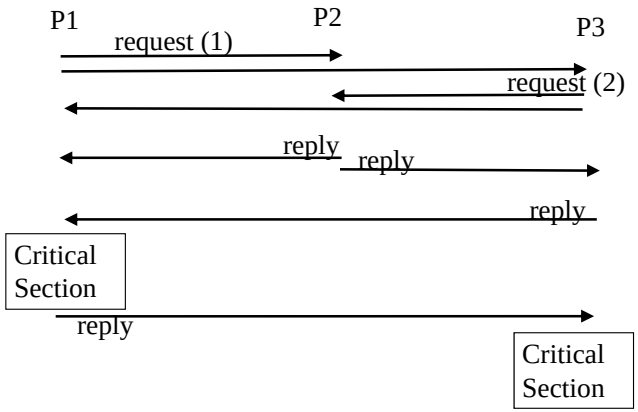
Advantages?

Disadvantages?

Mutual Exclusion – Decentralized Approach

- Requestor K:
  - Generate a TimeStamp TS_k.
  - Send request (K, TS_k) to all processes.
  - Wait for a reply from all processes.
  - Enter CS
- Process K receives a Request:
  - Defer reply if already in CS
  - Else if we don't want in, send reply.
  - (We want in) If $TS_r < TS_k$, send reply to R.
  - Else defer the reply.
- Leave CS, send reply to all deferred requests.

Example – Decentralized Approach



Advantages?

Disadvantages?

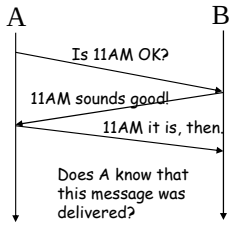
- Lost reply hangs entire system

Distributed control vs. central control

1. Distributed control is easier, and more fault tolerant than central control.
2. Distributed control is harder, and more fault tolerant than central control.
3. Distributed control is easier, but less fault tolerant than central control
4. Distributed control is harder, but less fault tolerant than central control

Generals coordinate with link failures

- Problem:
  - Two generals are on two separate mountains
  - Can communicate only via messengers; but messengers can get lost or captured by enemy
  - Goal is to coordinate their attack
    - If attack at different times → they loose !
    - If attack at the same time → they win !



Even if all previous messages get through, the generals still can't coordinate their actions, since the last message could be lost, always requiring another confirmation message.

General's coordination with link failures: Reductio

- Problem:
  - Take any exchange of messages that solves the general's coordination problem.
  - Take the last message m_n. Since m_n might be lost, but the algorithm still succeeds, it must not be necessary.
  - Repeat until no messages are exchanged.
  - No messages exchanged can't be a solution, so our assumption that we have an algorithm to solve the problem must be wrong.
- Distributed consensus in the presence of link failures is impossible.
  - That is why timeouts are so popular in distributed algorithms.
  - Success can be probable, just not guaranteed in bounded time.

Distributed concensus in the presence of link failures is

1. possible
2. not possible

Distributed Transactions -- The Problem

- How can we atomically update state on two different systems?
  - Generalization of the problem we discussed earlier !
- Examples:
  - Atomically move a file from server A to server B
  - Atomically move $100 from one bank to another
- Issues:
  - Messages exchanged by systems can be lost
  - Systems can crash
- Use messages and retries over an unreliable network to *synchronize* the actions of two machines?
- The two-phase commit protocol allows coordination under reasonable operating conditions.

Two-phase Commit Protocol: Phase 1

- Phase 1: Coordinator requests a transaction
  - Coordinator sends a REQUEST to all participants
    - Example: C → S1 : "delete foo from /"
      C → S2 : "add foo to /quux"
  - On receiving request, participants perform these actions:
    - Execute the transaction locally
    - Write VOTE_COMMIT or VOTE_ABORT to their local logs
    - Send VOTE_COMMIT or VOTE_ABORT to coordinator

Failure case	Success case
S1 decides OK; writes "rm /foo; VOTE_COMMIT" to log; and sends VOTE_COMMIT S2 has no space on disk; so rejects the transaction; writes and sends VOTE_ABORT	S1 and S2 decide OK and write updates and VOTE_COMMIT to log; send VOTE_COMMIT

Two-phase Commit Protocol: Phase 2

- Phase 2: Coordinator commits or aborts the transaction
  - Coordinator decides
    - Case 1: coordinator receives VOTE_ABORT or times-out → coordinator writes GLOBAL_ABORT to log and sends GLOBAL_ABORT to participants
    - Case 2: Coordinator receives VOTE_COMMIT from all participants → coordinator writes GLOBAL_COMMIT to log and sends GLOBAL_COMMIT to participants
  - Participants commit the transaction
    - On receiving a decision, participants write GLOBAL_COMMIT or GLOBAL_ABORT to log

Does Two-phase Commit work?

- Yes … can be proved formally
- Consider the following cases:
  - What if participant crashes during the request phase before writing anything to log?
    - On recovery, participant does nothing; coordinator will timeout and abort transaction; and retry!
  - What if coordinator crashes during phase 2?
    - Case 1: Log does not contain GLOBAL_* → send GLOBAL_ABORT to participants and retry
    - Case 2: Log contains GLOBAL_ABORT → send GLOBAL_ABORT to participants
    - Case 3: Log contains GLOBAL_COMMIT → send GLOBAL_COMMIT to participants

Limitations of Two-phase Commit

- What if the coordinator crashes during Phase 2 (before sending the decision) and does not wake up?
  - All participants block forever!
    (They may hold resources – eg. locks!)
- Possible solution:
  - Participant, on timing out, can make progress by asking other participants (if it knows their identity)
    - If any participant had heard GLOBAL_ABORT → abort
    - If any participant sent VOTE_ABORT → abort
    - If all participants sent VOTE_COMMIT but no one has heard GLOBAL_* → can we commit?
      - NO – the coordinator could have written GLOBAL_ABORT to its log (e.g., due to local error or a timeout)

Two-phase Commit: Summary

- Message complexity 3(N-1)
  - Request/Reply/Broadcast, from coordinator to all other nodes.
- When you need to coordinate a transaction across multiple machines, …
  - Use two-phase commit
    - For two-phase commit, identify circumstances where indefinite blocking can occur
    - Decide if the risk is acceptable
- If two-phase commit is not adequate, then …
    - Use advanced distributed coordination techniques
    - To learn more about such protocols, take a distributed computing course

Can the two phase commit protocol fail to terminate?

1. Yes
2. No

Who's in charge? Let's have an Election.

Many algorithms require a coordinator. What happens when the coordinator dies (or at startup)?

- Bully algorithm

Bully Algorithm

- Assumptions
  - Processes are numbered (otherwise impossible).
  - Using process numbers does not cause unfairness.
- Algorithm idea
  - If leader is not heard from in a while, assume s/he crashed.
  - Leader will be remaining process with highest id.
  - Processes who think they are leader-worthy will broadcast that information.
  - During this "election campaign" processes who are near the top see if the process trying to grab power crashes (as evidenced by lack of message in timeout interval).
  - At end of time interval, if alpha-process has not heard from rivals, assumes s/he has won.
  - If former alpha-process arises from dead, s/he bullies their way to the top. (Invariant: highest # process rules)

Bully Algorithm Details

- Bully Algorithm details
  - Algorithm starts with Pi broadcasting its desire to become leader. Pi waits T seconds before declaring victory.
  - If, during this time, Pi hears from Pj, j>i, Pi waits another U seconds before trying to become leader again. U?
    - U ~ 2T, or T + time to broadcast new leader
  - If not, when Pi hears from only Pj, j<i, and T seconds have expired, then Pi broadcasts that it is the new leader.
  - If Pi hears from Pj, j<i that Pj is the new leader, then Pi starts the algorithm to elect itself (Pi is a bully).
  - If Pi hears from Pj, j>i that Pj is the leader, it records that fact.

In the bully algorithm can there every be a point where the highest number process is not the leader?

1. Yes
2. No

Byzantine Agreement Problem

- N Byzantine generals want to coordinate an attack.
  - Each general is on his/her own hill.
  - Generals can communicate by messenger, and messengers are reliable (soldier can be delayed, but there is always another foot soldier).
  - There might be a traitor among the generals.
- Goal: In the presence of less than or equal to f traitors, can the N-f loyal Generals coordinate an attack?
  - Yes, if $N \geq 3*f+1$
  - Number of messages ~ $(f+1)*N^2$
  - f+1 rounds
  - (Restricted form where traitorous generals can't lie about what other generals say does not have N bound)

Byzantine Agreement Example

- N = 4 m = 1
- Round 1: Each process Pi broadcasts its value Vi.
  - E.g., Po hears (10, 45, 74, 88)
  - Vo = 10
- Round 2: Each process Pi broadcasts the vector of values, Vj, j!=i, that it received in the first round.
  - E.g., Po hears from P1 (10,45,74,88),
  - from P2 (10,66,74,88) [P2 or P1 bad]
  - from P3 (10,45,74,88)
- Can take all values and vote. Majority wins. Need enough virtuous generals to make majority count.
- Don't know who virtuous generals are, just know that they swamp the bad guys.

Byzantine Agreement Danger

- N = 3 m = 1
- Round 1: Each process Pi broadcasts its value Vi.
  - E.g., Po hears (10, 45, 75) [P2 is lying]
- Round 2: Each process Pi broadcasts the vector of values, Vj, j!=i, that it received in the first round.
  - E.g., Po hears from P1 (10,45,76),
  - from P2 (10,45,75)
  - Either
    - R1: P2 told Po 75
    - R1: P2 told P1 76
    - P1 is trustworthy, P2 lies about P1
  - OR
    - R1: P2 told Po 75
    - R1: P2 told P1 75
    - P2 is trustworthy, P1 lies about P2 in R2
  - Po can't choose between 75 and 76, even if P1 non-faulty

Byzantine fault tolerant algorithms tend to run quickly.

1. Yes
2. No