Distributed Software Development

Consensus

Chris Brooks

Department of Computer Science
University of San Francisco

7-0: Consensus and agreement

- A fundamental problem in distributed systems is getting a set of processes or nodes to agree on one or more values.
  - Is a procedure continuing or aborted?
  - What value is stored in a distributed database?
  - Which process is serving as coordinator?
  - Has a node failed?
- There are a set of related problems that require a set of processes to coordinate their states or actions.

7-1: Coordination via email

- An example:
  - Two people (A and B) want to meet at dusk tomorrow evening at a local hangout.
  - Each wants to show up only if the other one will be there.
  - They can send email to each other, but email may not arrive.
  - Can either one guarantee that the other will be there?

7-2: Failure models

- We'll want to distinguish what sorts of failures these algorithms can tolerate.
- No failure
  - Some of the algorithms we'll see can't tolerate a failure.
- Crash failure
  - This means that a node stops working and fails to respond to all messages.
- Byzantine failure
  - A node can exhibit arbitrary behavior.
  - This makes things pretty hard for us ...

7-3: Failure detection

- How can we detect whether a failure has happened?
- A simple method:
  - Every t seconds, each process sends an "I am alive" message to all other processes.
  - Process p knows that process q is either *unsuspected*, *suspected*, or *failed*
- If p sees q's message, it knows q is alive, and sets its status to unsuspected.
- What if it doesn't receive a message?

7-4: Failure detection

- Depends on our communication model.
- Synchronous communication: if after d seconds (where d is the maximum delay in message delivery) we haven't received a message from p, p has failed.
- Ansychronous or unreilable communication: if the message is not received, we can say that p is suspected of failure.

7-5: Failure detection

- Other problems:
  - What if d is fairly large?
  - We can think processes are still running that have in fact crashed.
- This is what's called an *unreliable* failure detector.
- It will make mistakes, but, given enough information, it may still be of use.
- Can provide hints and partial information.
- As we look at different algorithms, we'll need to think about whether we can detect that a process has failed.

7-6: Multicast: a brief digression

- The Couloris chapter talks quite a bit about how to achive different properties with multicast communication.
  - Reliable multicast
  - Ordered multicast
    - FIFO ordering
    - Total ordering
    - Causal ordering
- The punchline: Totally ordered multicast is equivalent to the consensus problem.
- Implementing one or more of these on top of IP multicast could be a cool final project.

7-7: What is multicast?

- Consider that a process needs to send a message to a *group* of other processes.
- It could:
  - Send a point-to-point message to every other process.
    - Inefficient, plus need to know all other processes in group.
  - Broadcast to all processes in subnet.
    - Wasteful, won't work in wide-area network.
- Multicast allows the process to do a single send. Packet is delivered to all members of the group.

7-8: Multicast groups

- Notice that multicast is a packet-oriented communication.
  - Same send/receive semantics as UDP
- A process joins a multicast group (designated by an IP address)
- It then receives all messages sent to that IP address.
- Groups can be closed or open.
- Multicast can be effectively used to do shared whiteboards, video or audio conferencing, or to broadcast speeches or presentations.
  - Middleware needed to provide ordering.

7-9: Mutual exclusion

- Mutual exclusion is a familiar problem from operating systems.
  - There is some resource that is shared by several processes.
  - Only one process can use the resource at a time.
  - Shared file, database, communications medium
- Processes request to enter their *critical section*, then enter, then exit.
- In a centralized system, this can be negotiated with shared objects. (locks or mutexes).
- Distributed systems rely only on message passing!

7-10: Mutual exclusion

- Our goals for mutual exclusion:
  - safety: Only one process uses the resource at a time.
  - liveness: everyone eventually gets a turn.
    - This implies no deadlock or starvation.
  - ordering: if process i's request to enter its CS happens-before (in the causal sense) process j's, then process i should enter first.

7-11: Mutual exclusion: centralized server

- One solution is to use a centralized server to manage access.
- To enter the critical section, a process sends a request to the server.
  - If no one is in a crtical section, the server returns a token. When the process exits the critical section, the token is returned.
  - If someone already has the token, the request is queued.
- Requests are serviced in FIFO order.

7-12: Mutual exclusion: centralized server

- If no failures occur, this ensures safety and liveness.
- Ordering is not satisfied.
- Central server provides a bottleneck and a single point of failure.

7-13: Mutual exclusion: token ring

- Rather than dedicating a server, processes are logically arranged in a ring.
  - This may not have anything to do with network topology; we just assign an order.
- The token is passed clockwise (for example) around the ring.
- When the token is received, a process can enter its critical section.
- If the token is not needed, it is immediately passed on.

7-14: Mutual exclusion: token ring

- Achieves liveness and safety, but not ordering.
- Also can use a lot of bandwidth when no process needs the resource.
- Also can't handle failure.
  - What if the process with the token fails?
  - If we can detect failure, we can generate a new token.
  - Leader election can deal with this.

7-15: Mutual exclusion: multicast

- Assume each process has a distinct identifier
- Assume each process can keep a logical clock.
- A process has three states: Released, waiting, held.
- When a process p wants to enter the critical section:
  - p sets its state to waiting
  - p sends a message to all other processes containing its ID and logical timestamp.
  - Once all other processes respond, p can enter.

7-16: Mutual exclusion: multicast

- When a message is received:
  - If state is held, queue message
  - If state is waiting and timestamp is after local timestamp, queue message.
  - Else, reply immediately.
- When exiting the critical section:
  - set state to released
  - reply to queued requests.