CptS 464/564
Lecture 12

30th September 2004

Continuing the discussion of Coordination (2.8)

Last time we discussed coordination in centralized systems – locks, semaphores and monitors. We continue today with a look at approaches to coordination problems in distributed systems.

Mutual exclusion in decentralized systems

Coordinator approach - Lock Server. Figure 2.25

Problems – Lock server is a single point of failure. Failure of client holding the lock will also cause problems.

Distributed mutual exclusion: naive approach – ask everyone for permission; when permission is granted by all processes, enter the critical region. Problems? lack of coordination leads to several processes each holding "part of" the lock, so no process can make progress.

Leader election as an optimization of distributed mutual exclusion: look at on your own.

Deadlock

A *deadlock* occurs when two or more processes are waiting for each other and no progress can be made. Conditions for deadlock:

- mutual exclusion – one process can do something that keeps another from making progress; typically, holding some resource
- hold-and-wait – it is possible to wait for a resource while holding another resource
- non-preemption – one process can't force another to let go of a resource
- circular waiting: there exists a chain of process in a hold-wait cycle

Eliminating Deadlock

Elimate one of the necessary conditions – not so easy. First two are fundamental for many algorithms and data structures.

Allow pre-emption: greatly complicates application code – may introduce asynchronous interruption into otherwise single-threaded code.

Prevent circular waiting – always acquire resources in the same order (define a total order on resources)

Deadlock detection: detect cyclical waiting and kill a process (a brutal form of pre-emption!)

Deadlock avoidance: don't acquire a lock that would cause a deadlock.

- Note 1: deadlock is a *stable property* of a system: once it occurs it remains. Therefore, if we ever capture a consistent state of the system that shows it in deadlock, we know that it has persisted through subsequent state changes and therefore still has to be dealt with. Potential deadlock (i.e. deadlock that will occur if one more lock is acquired, is not a stable property, and therefore harder to detect accurately.
- Note 2: both of the last two require capturing a consistent global state of a distributed computation. Leading us into our next topic.

Consistency (Sec. 2.9 - and also 1.4.3)

In my opinion the authors' given motivation for calling this section "consistency", in the intro to the section, is incomplete, though it is not a bad title. The consistency we are *first* concerned with here is that required for obtaining an accurate understanding of the global state of a distributed system: an understanding that captures our notion that *state* explains the future behavior of a system. In short, what is a *consistent global state?*

We explore the quesiton from two perspectives. First, as an outside observer of a system, what does a consistent global state look like? Second, how can a participant (inside) a distributed system determine a consistent global state? The answer to the second question builds on the first.

Once we have an answer to the second question, we will be able to use it to build distributed algorithms solving useful problems such as consensus and deadlock detection.

The second sense of consistency, (which is more consistent :-) with the author's intro to the section) concerns specification and design issues: what is the requirement for different nodes in a distributed system to maintain the same (consistent) state and provide the same (consistent) replies to clients? A broad range of answers to this question is possible: which one is chosen depends on the application and people's expecations of its behavior.

Examples:

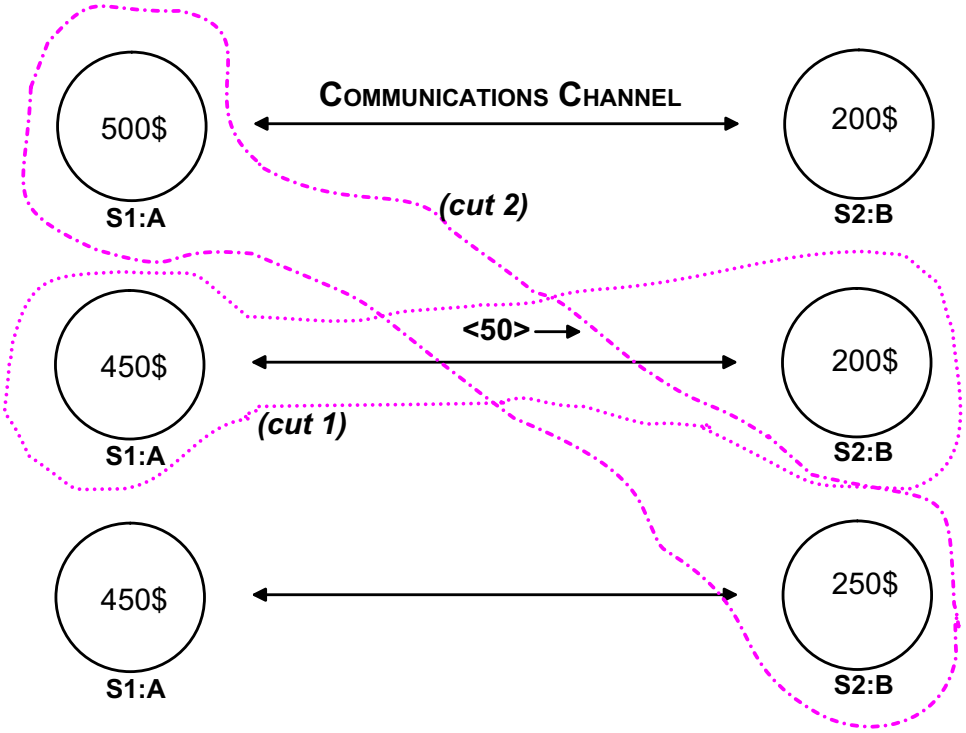
- strongly-consistent replicated database - state is everywhere the same; replicas give the same answer at any give "time".

- eventually-consistent replicated database: as long as updates are occurring, replicas may give different answers; if updating stops, after a sufficiently long time, replicas will give consistent answers

The intro also suggests that "consistent" is used in the sense of meaning an implementation is correct with respect to its specification.

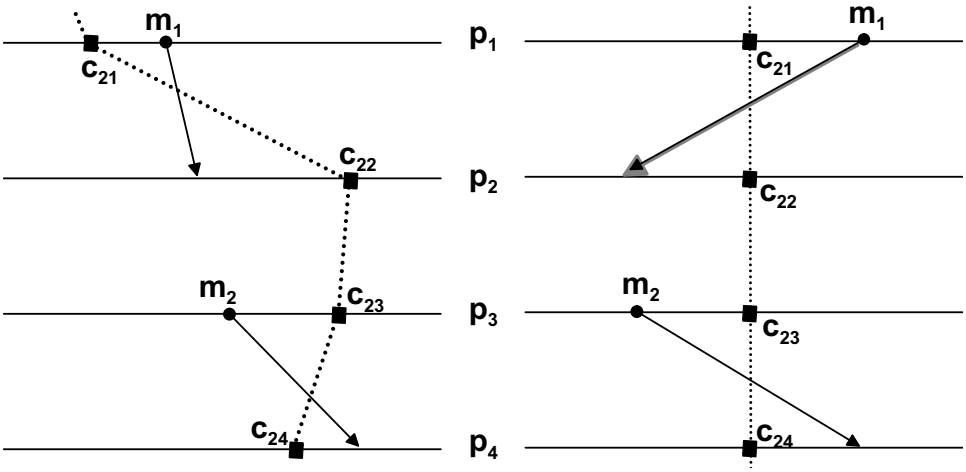
Achieving a consistent picture of the state of a DS

Refer to Figure 2.28 (Redraw in standard form for lecture). Banking example: How much money is in the system? Cut 1- $650; Cut 2: $750. Correct answer: $700.



Key notion is that of a *consistent cut:* a collection of states, one from each process in the system, in which all messages that have been received in the state of some process have been sent in the state of another. It is maybe easier to see what is an inconsistent cut: a set of states, one from each process in the system, in which some message has been received but not sent. A system could have beeen in the state captured by a consistent cut; it most certainly could not have been in the state represented by an inconsistent cut. We also have the notion of a strongly consistent cut: one in which is consistent and in which all messages that have been sent have also been received.

Refer to Figure 2.29: What feature of this diagram tells us that we are seeing an inconsistent cut? If that message is removed, what feature tells us we are seeing a cut that is not strongly consistent?

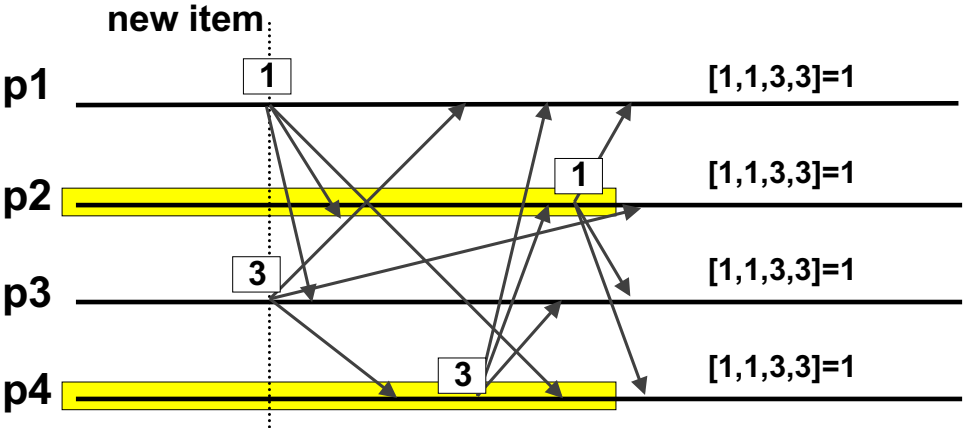


(Not covering, not testing Lamport/Chandy distributed snapshot protocol)

Simple Distributed Consensus

Idea: a collection of processes must agree on a single value. Which one machine will pick up the package? Free machines say "I will pick it up"; busy machines wait for either becoming free, in which case they say "I (Sam) will pick it up" or receiving a proposal from someone else, in which case they say "Someone else will pick it up." Notice–each machine makes one proposal sent to all others – it never changes its mind. Once a proposal is received from everyone, use a deterministic rule (e.g. earliest alphabetical name) to decide who picks up the package. Everyone has received the same messages so all decide the same answer.

Refer to Fig. 2.30 for example.



Notice that the algorithm won't work if processes fail or messages are lost. Thus the "simple" in the title of this section. Overcoming failures is the subject of Fault Tolerance which is the subject of CptS 562.

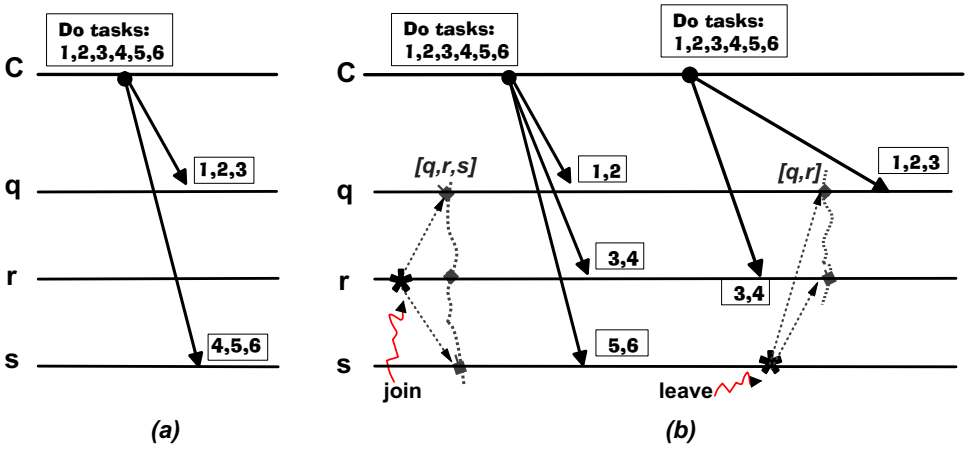
Simple Agreement on Membership

Recall our earlier observation that the totally ordered broadcast algorithm works only when current collection of processes is accurately known to all processes. The distributed consensus algorithm above also requires this knowledge. The problem is generally known as "agreement on membership."

A membership service is an algorithm for providing processes a consistent *view* of the membership of a group. One approach: ordered views: the processes see an evolving sequence of states (views). They all agree on the membership of each view, V^i, V^{i+1},V^{i+2}... The next view is determined by the processes in the current view using a deterministic algorithm, much as in the Simple Consensus example.

Again, failure of the algorithm ensues if processes fail or messages are lost. Even in the absence of failure, though, this simple view

Example: collection of servers divides batches of work amongst themselves equally.

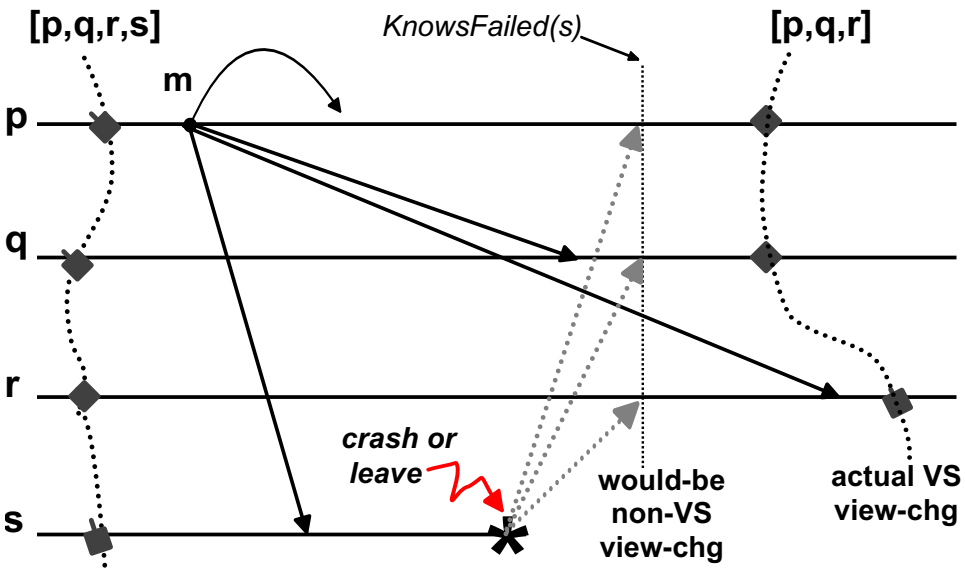


Refer to Figure 2.31(a).

Servers q and s are currently in the group. They divide the work batch equally using a deterministic (and locally executable) algorithm. Server r joins the group. The three processes together process the next batch (Fig. 2.31(b)). Show the view update. Server r leaves the group. Show the work batch messages arriving in different views.

Solution: *View Synchrony* or *Virtual Synchrony:* ensure that delivery of every message occurs in the same view at all processes. Solution: consensus – proposal for new view contains not only the proposed new members but also the messages that make up the current view. This may result in delaying messages.

Refer to Fig. 2.32



Skipping Atomic Broadcast and Replica Determinism (2.9.5-6) for now

Partitions (2.9.7)

When some portion of a DS is not able to communicate with some other portion, we say that the DS has *partitioned*, and we call each of the pieces that remain in communication a *partition.* In a partitioned DS, the problem is that the different partitions will evolve independently (*diverge)*, so that when the partition is repaired, it is completely unclear how to construct a new state that represents what has happened in both pieces.

Since partitioning is inevitable, the system design has to dictate what will happen when it occurs. One commonly used approach is the notion of a *primary partition.* If the system partitions, at most one partition is allowed to continue changing its state. Each node has to be able to decide locally whether it is part of this priviledged partition. Two ways that might be used: *priviledged node* - a particular node is designated as the most important; the partition containing this node is the primary partition; *majority partition* - only a partition containing a majority of the nodes making up the DS can proceed.

Both have flaws: if the priviledged node goes down, the system cannot progress; similarly, if no partition contains a majority, the system cannot progress.

Alternative: use a weaker notion of consistency, if possible, allows partitioned systems to make progress. This is especially helpful in systems where partitioning is frequent and expected (mobile computing, e.g.).

Skip section 2.10

Atomicity

The key notion to take from the Atomicity paradigm is that of a collection of actions, whose results either all occur or none occur, with none of the intermediate steps being visible.

Example: buying a plane ticket for a multi-hop trip. You wouldn't want to be stuck with tickets for only some of the legs of your trip.

Achieving atomicity for a series of operations requires:

- the willingness to accept "none occur" as an outcome
- the ability to *roll back* partially completed sequences as if they had never occurred
- the ability to hide intermediate states from other processes
- the ability to take a single action that *irrevocably commits* all the prior actions; usually, this idea incorporates the notion of being able to recover after hardware failure, so stable storage and logging are key pieces of implementing transactional systems.
- in a distributed system, all of these requirements still exist, of which the last is probably the most difficult to achieve; two-phase commit is one solution (described in the book), but others may exhibit more availability in the face of unreliable networks.

Chapter 2 Summary

The DS paradigms provide us a vocabulary and common set of techniques that occur frequently in talking about and building DS. Next time we will start Chapt. 3 wherein we look more at how these ideas fit together.