CptS 464/564
Lecture 14

12th October 2004

Housekeeping

Today: hand in homework 1 (grad students have 'til Tuesday for the research-based questions)

Reminder: Mid-term exam is next Thursday, October 14. Tuesday I will be teaching from TC and will have a project workshop there after class.

MessengerName example: Harald has written a demonstration of using the CORBA name service. I will put it in the orbacus/cs564 directory this evening.

Distributed Systems Models

Why models (taken from Mullender, S. (ed.), *Distributed Systems 2nd ed., chapter 2, "What Good are Models and What Models are Good? by F. Schneider*).

- Refining intuition - two approaches to developing intution: experimental observation; modeling and analysis. In modeling we "simplify the object of study and postulate rules to define its behavior." Tension between "practitioners" and "theoreticians". Problems: theory of the wrong thing (from practitioners point of view), incorrect inference and generalization (from the theoreticians point of view).

What is a model: attributes and rules describing their interactions. What is a good model: *accuracy* and *tractability.* Logical models vs. simulation models. Logical models allow reasoning about a system, simulation models allow observation of a simpler system than the real one perhaps making inference easier.

- infrastructure – local vs wide area network, what

DS Frameworks

- Note this matter of terminological confusion: we're using the term *frameworks* here to mean logical and conceptual frameworks. How do we organize our thinking about these matters. The term *framework* is also applied to specific software systems that embody (partial) design solutions in a particular space. For example, CORBA would be called a low-level framework for distributed computing.
- Infrastructure - what are the resources; how should they be organized? Why?
- Semantics - which paradigms are appropriate in a given situation, in order to achieve the architectural objectives
- Models - for organizing one's thinking about DS activities and services
- Distribution of information repositories
- Access paradigms - how do users access services?

Infrastructure

Infrastructure includes hardware, networks, OS support. Networking infrastructure is perhaps the most important for DS: important considerations

- is it a private network (within a single organization)? (intranet)
- does it use a public network such as the world-wide Internet?
- does it hook two (or more) intranets together more or less privately (extranet)
- Security infrastructure

Time: Asynchronous models

Diagram: model/platform/framework layer; application layer. An asynchronous platform makes no guarantees about how long anything takes.

As we've previously mentioned, in a fully asynchronous model, there is no notion of time durations. We have notions of before, after, and concurrent with, but we don't/can't measure delays. Consequently, processing and message delivery delays are unbounded or unknown; rate of drift of clocks is unknown as is the difference between clocks. (Indeed, in a fully asynchronous system, we would have to be satisfied with logical clocks, as described in Chapt. 2!)

An application running on an asynchronous platform will have to be designed relying on only very weak guarantees such as:

- messages are eventually delivered (but there is no bound on how long they take)

- execution eventually terminates
- failures are eventually detected

These guarantees are so weak that it becomes impossible to implement some desirable behaviors:

- Fischer, Lynch and Paterson, in 1985, showed that distributed consensus is *impossible* if nodes can fail
- deterministic reconfiguration of a system (e.g. primary partition technique) may not work

The underlying problem: it is impossible to tell the difference between a *failed* participant and one the is merely *slow.*

Bottom line: as an implementor of a framework you will like a requirement to build a fully asynchronous system. As an implementor of DA you will dislike being told it is to be implemented on an asynchronous framework.

Time: Synchronous Models

People's expectations for systems usually include some notion of a bound on how long activities should take: if you order a burger it should take at most a few minutes; if you order pizza it should take at most 20 mniutes. It would not be acceptable for the restaurant to say, you'll get it eventually – that might mean a year from now.

Synchronous and partially synchronous frameworks are required in order to address expectations of bounded delay. In a fully synchronous system, we can bound the delays on processing and message delivery, and also on differences between clocks. Building and maintaining a fully synchronous framework can be very difficult: among other things, you can't depend on synchronous behavior of the environment, and you have to strictly control how much load is placed on the system (otherwise, queueing delays can overwhelm all the care you've put into implementing processing and message delivery delays). The problem is that from your client's perspective having requests for service repeatedly rejected is no better than dealing with a fully asynchronous system! (I.e. there is no bound on how long I have to wait to get my request accepted into the system.) Diagram: finite capacity server with bounded delays – can't be maintained in the face of arbitrarily large load.

As a practical matter, most systems are built assuming neither fully synchronous nor fully asynchronous platforms. The result is a fairly muddled picture of what is being guaranteed about their behavior.

Failures

- Fail-stop: processors fail by halting and never restart; failure is detectable by other processors
- Crash: as above but the failure may not be detectable by other processors
- Crash+Link: processors fail as above; links fail by losing but not corrupting, delaying, duplicating or reordering messages
- Receive omission: processor fails by not receiving some messages sent to it or by halting and never restarting
- Send omission: processor fails by not actually sending some of the messages it is supposed to send (or by halting...)
- General omission: Receive omission + Send omission
- Byzantine: arbitrary behavior

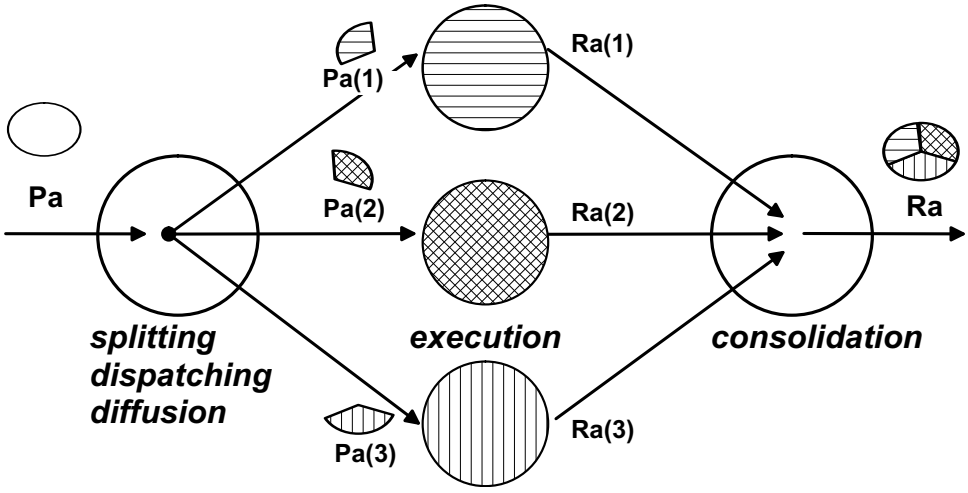
Example using a logical failure model to discover something interesting. Suppose that in a system of two processors, A and B, connected by a link neither process can fail but the link may omit messages. Suppose for this system we want to devise a protocol such that either both A and B do action a or both A and B do action b or both A and B do neither action. In this model the problem has no solution. Proof by contradiction:

- Assume there are protocols that solve the problem. Select one of these protocols, P, that solves the problem in the fewest rounds.
- In this protocol there is a last message, *m*, sent. Assume, without loss of generality, that it is sent by A. Observe that A's action cannot depend on whether *m* is received by B (because *m* is the last message A can never learn whether it is received). Similarly, B's action cannot depend on whether *it* receives *m* – because it must take the same action regardless of whether it receives the message. *m* is therefore superfluous so there must be a shorter protocol that solves the problem contradicting the assumption that P was a shortest protocol.

Design decomposition for DS (3.5) Classes of distributed activities

Several classes of activities and interactions occur repeatedly in the design of distributed systems. We will look here at three of them: coordination, sharing and replication. There are lots of variants of each and they can be connected in arbitrary ways, giving a very rich space of possible designs.

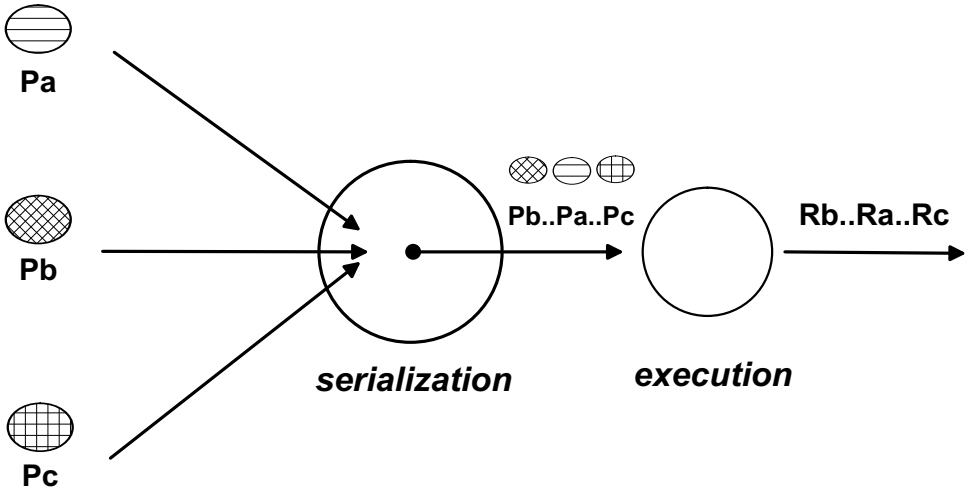
Coordination



Coordination (Fig. 3.3) activities organize and control the work done in a distributed system: they get the work to the sites that make up the DS and form a final result from the partial results computed at the several sites. Steps of coordination include

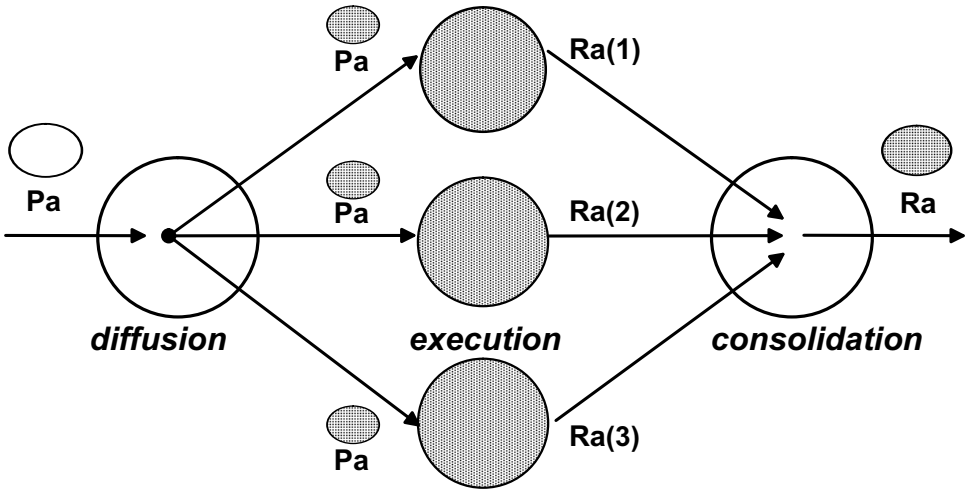
- Splitting: subdividing a main task into subtasks
- Dispatching: allocating the subtasks to the sites
- Diffusion: conveying the subtasks to the sites
- (Execution): the sites process their subtasks
- Consolidation: collect, combine, and send onward the results of the subtasks

Sharing



Sharing activities (Fig. 3.4) include the steps needed to ensure correct/consistent use of shared resources. One kind of sharing: *serialize* subtasks accessing a resources. That is, subtasks access the resource one at a time. What serial order is appropriate? It depends on the problem: FIFO, causal, optimized by commutation, are all possible.

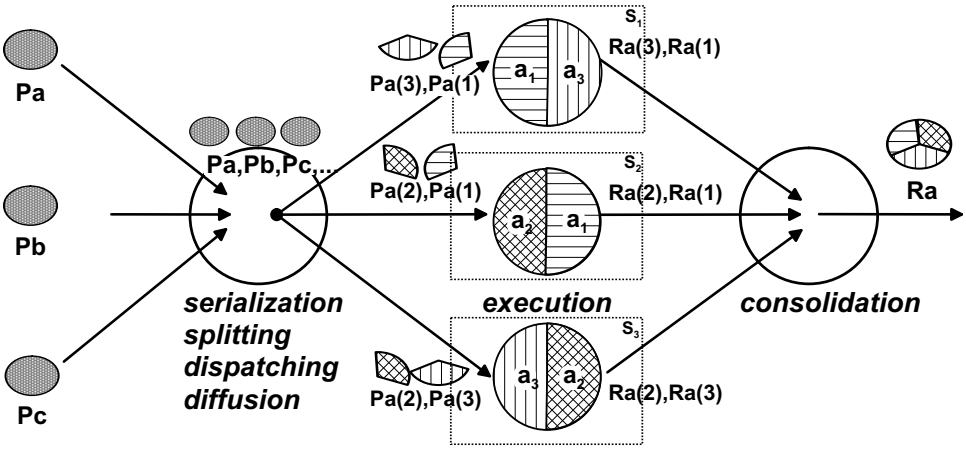
Replication



Much the same set of steps as coordination, but with a different purpose: instead of splitting a task into subtasks, we now wish to execute a task (subtask) at multiple sites. Why? possibilities include availability, fault tolerance, increased performance. The steps are

- (dispatching – not mentioned in the text, but possible): decide which sites will participate
- diffusion
- execution: since all sites have the same task, depending on our purpose, all of the sites may process it (active replicas) or some may hold it to process only if an active site fails (passive replicas).
- consolidation: combine the results; remember that all sites executed the same task. Thus consolidation for various purposes might involve: taking the first response, taking a majority response.

Combined activities



In section 3.2 there were something like 8 dimensions, many with more than 2 choices. But not all of the combinations make sense. In section 3.5 we saw three fundamental activities that occur in DS. But let's pause and reflect a bit on why a distributed system might be desired. Exercise: reflect on which of the activities and tradeoffs would be relevant to each of these motivations.

- Availability and performance? remember that components of DS exhibit independent failures. While this complicates our lives compared with the centralized case (e.g. partitiioning, partial failures, etc.) it is also an opportunity to increase availability of the system (for some of the clients, some of the time). Likewise, distributed clients of a centralized system see different performance; if we distribute the services as well as the clients perhaps we can provide better and more uniform performance for the clients.
- Modularity? although a centralized, integrated system may be easier to manage in a static environment, DS make it easier to deal with growth and organizational change.

- To reflect a decentralized organization? if pieces of the organization have a lot of autonomy a distributed system may support it better in the way that it works
- For security? related to availability–don't put all your eggs in one basket; fragment data so that access to multiple locations is required to recreate it; redundancy–voting by servers on the correct answer. Draw–secret sharing; voting