CptS 464/564
Lecture 14

3rd March 2003

Last time we looked at a number of dimensions on which DS designs could vary.

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.

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.

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

Figure 3.6.

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