CptS 464/564
Lecture 17

2nd November 2004

Summary

- Finishing chapter 3, in particular section 3.9

RPC (3.6)

Most of this material should be familiar based on earlier discussions about CORBA, but there are a few architectural issues of interest that are addressed here, in particular the use of multiple threads of control.

The first case to consider is when in handling a request a server might be delayed by something like local slow operations (e.g. disk reads and writes). If the server is single-threaded this will keep any other requests from being processed. One way to overcome this is to allow the server to create a thread for each incoming request. This incurs a cost for creating a thread each time but by processing requests concurrently it make better use of the compuational resources. Optimizations such as *thread pools* are often used to eliminate the cost of forking a new thread for each request. The choice to use multiple thread also requires that the server's design take account of the synchronization requirements between request which may have to do with protecting local resources from inconsistent updates or it may be imposed by the need to respect causality between incoming requests.

Another approach to improving utilization of a server is to use a so-called *event-driven* architecture. This is a single-threaded model using non-blocking I/O. Rather than relying on OS or language support for threads, event-driven applications explicitly manage concurrent activity by breaking down their work into small chunks that are invoked from the *event loop.* The event loop looks like:

```
while(1) {
   e = waitForEvent();
   process(e);
}
```

waitForEvent() produces events corresponding to incoming requests as well as completion of work fragments. process(e) may involve finishing a client request, or doing some processing on it followed by starting a time-consuming external operation.

Using a multi-threaded or event-driven approach is especially important if requests made to one server may result in further calls on other servers for two reasons: the first is the obvious one concerning performance; the second is that deadlock is possible if single-threaded servers can call each other.

Notice that RPC exhibits temporal and spatial coupling: both parties must be present at the same time, and their is a tight binding of the client and server: the client must figure out where the server is before starting an interaction and the server sends its reply to a specific client.

Group Communications (3.7) and Distributed Shared Memory (3.8)

We are skipping these sections. Group communications is pretty cool – see papers on Isis, Horus, and Ensemble from Birman *et al.* at Cornell, for example.

DSM is a concept more appropriate for high-performance computing where the idea is to put a lot of computational power as close together as possible in order to solve big computational problems. The analogy is that a RPC is to local procedure call as remote memory reference in DSM is to local memory reference. We have thought about the implications of distribution for procedure calls. Now think about the ubiquity of memory reference in programs and consider the implications of latency, failures, *etc.*

Message Buses also known as Message Oriented Middleware (MOM) (3.9)

This is good background for the third project and we will talk about it in some detail.

The idea of a message bus is that producers/senders and consumers/receivers are decoupled in time and space: messages may be sent at one time and delivered later (*temporal decoupling)*; also senders and receivers need to know less about one another's identities (*spatial decoupling)*. (Notice that group communications/multicast mechanisms provide some aspect of this latter property). (Makes it easier to configure the system.)

(Use the figure from TvS on the ppt slide here)

One use of the message bus idea is in building *publish-subscribe* systems. Publishers (producers/senders) send messages to the bus. Subscribers indicate to the bus their interest in a certain kind of message. The bus delivers the requested messages to subscribers. Publishers and subscribers need not know one another's identities.

For programmers in conventional languages MOM is less transparent than RPC, but programmers used to some styles of local programming (highly concurrent using channel-based communications) find it quite natural.

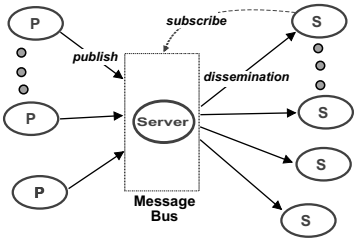
Volatile vs. persistent buses (*resilience)*: are messages delivered to subscribers that weren't around to receive them when sent? (Temporal decoupling). Yes – persistent bus; no – volatile bus.

Local vs. global buses (*scope)*

Subscribe to all vs. subscribe by subject: CORBA Event Service vs. CORBA Notification Service. This is a fuzzy distinction because in some sense having different buses for different subjects suffices. But subject-based subscription with pattern matching is more flexible and requires less preconfiguration.

Netnews newsgroup as an example of a persistent message bus: messages posted to the newsgroup, not individual subscribers; subscriber doesn't have to be available at the time a message is posted.

We will explore in project 3 some of the implications of *push* vs. *pull* approaches to MOM mentioned in the book.



Implementation of MOM: these days, almost all middleware and distributed applications are built, at some level, over the Berkeley sockets interface, whether using Unix-like OSs or Windows-derived OSs. Socket programming uses a particular set of system calls (or library calls): socket, bind, listen, accept, connect, send, receive, read, write, close. (Socket programming is taught in CptS 455).

MOM repackages the socket functionality so that it is more application-friendly (but less capable!).

A very useful add-on to the MOM concept is the idea of Message Brokers. MBs address one of the problems caused by the evolution of systems in business. Essentially, systems accrete. New functionality is built on old functionality by making new applications talk to old ones. However message formats may be different between old and new, yet both must ultimately "understand" each others' messages. One approach is to try to standardize on a common message format but this can severely limit the functionality (and doesn't really work in the long term).

Another approach is to provide message translation services, also known as Message Brokers. A message broker can be integrated with a MOM system or be a stand-alone service.

(figure from second ppt slide).