Lecture 19
Concurrent Programming

21st November 2002

Linda Barriers revisited

Thanks to Bedirhan and Ryan we have two examples of resusable barriers in Linda.

Bedirhan offers a solution that uses a coordinator process:

```
Initialize the barrier and execute the coordinator:
OUT("barrier", 0)

Coordinator will execute:
OUT("coord", 0)
IN("coord", n)
IN("barrier", n) ---> we need this
OUT("barrier", 0)

Each time a process arrives at the barrier it executes;
IN("barrier", ?counter)
OUT("barrier", counter+1)
RD("barrier", n)
IN("coord", ?counter)
OUT("coord", counter+1)
```

Ryan's solution uses two barriers

```
Init:
OUT ("Barrier", 0)

1 Process (Arbitrary):
IN("Barrier", ?ct)
OUT("Barrier", ct+1)
```

```
IN("Barrier", N)
OUT("Barrier2", N-1)
IN("Barrier2", 0) // Waits for count down on second barrier
OUT("Barrier", 0) // Restarts the first Barrier

Other Processes:
IN("Barrier", ?ct)
OUT("Barrier", ct+1)
IN("Barrier2", ct)
OUT("Barrier2", ct-1)
```

Or with two special processes you can decrement a single counter back to 0:

```
Init:
OUT(“barrierbegin”)

First special process:
IN (“barrierbegin”)
OUT(“Barrier”, 1)
IN (“Barrier”, 0)
OUT(“barrierbegin”)

Second special process:
IN (“Barrier”, N-1)
OUT (“Barrier”, N-2)

Other processes:
IN (“Barrier”, ?ct)
OUT(“Barrier”, ct+1)
IN (“Barrier”, ct)
OUT(“Barrier”, ct-1)
```

MPI

MPI (Message Passing Interface) is a standard library for message-passing programming implemented on many systems. You may have heard people talking about Beowulf clusters – many commodity computers hooked up with a fast network to exploit parallel processing. They are often programmed using the MPI library. MPI is more targeted at exploiting parallelism than toward structuring programs to manage concurrency.

The key concurrency operations in MPI are MPI_Send and MPI_Receive. Messages are sent to and received from specific processes – though receive allows the option to receive from any process). There is also a notion of a process group and the ability to broadcast a message to all members of one's group. Whether

Send is synchronous or asynchronous is left unspecified, though most implementations provide asynchronous send (or a least the ability for a programmer to create one).

MPI also includes primitives intended to ease distributing work to other processes and gathering results from them. The MPI_Scatter operation sends each element of an array to a separate process. MPI_Gather collects results by storing a message from each other process into an array.

Finally, MPI provides a *reduce* operation that allows a process to gather and combine results from other processes in a single operation. (The notion of reduction: given a list of values and a binary operator, insert the operator between each pair of elements of the list: reduce([1,2,3,4,5],+)==15, reduce([1,2,3,4,5],min)==1, reduce([1,2,3,4,5], *)==120, etc.)

Advertisement: next semester Professor Murali Medidi will be teaching CptS 550, Parallel Computation. I expect that there will be much more about MPI in that class, including some opportunity to program with it.

RPC - Remote Procedure Call

It seems a little odd to discuss RPC as a primitive concurrent programming mechanism – we'll discover that we really require other concurrency mechanisms to make it meaningful. However, RPC provides a reference point for discussing rendezvous, next.

RPC is best thought of using a client-server model. Client programs call procedures (remotely) on servers. The idea is that from the client perspective such procedure calls will very closely resemble normal procedure calls within the client's address space, but in fact the procedure will be executed by the server, in a different address space or on a different machine.

There are a number of concepts entwined with RPC:

- first, servers: how are servers identified. The answer depends on the RPC system in use. It might be as simple as an IP address and port number, or servers may have identities that have to be looked up in some directory service and translated to contact information. Often a preparatory phase is performed where the client obtains a *handle* for the server and uses it in subsequent operations. Programming languages with object support are very handy here: a collection of remote procedures can be represented by a single object along with its methods. The object manages connection information internally, so in the client program remote method invocations and local method invocations are indistinguishable.
- second, synchronization: a classical RPC behaves just like normal a procedure call: the calling program does not pass the point of the call until the call returns. In message passing terms, the client *sends* its request and *receives* its reply in a single operation that looks like a procedure call.

- third, argument passing: the server needs to receive the arguments passed by the caller. Since the client and server are do not normally share an address space, this means that RPC is usually implemented with *call-by-value* semantics, and the arguments are copied into the RPC call message that is sent to the server. In object-oriented systems a reference to an object may be passed instead – in which case the server may make additional method calls on what, to it, are now remote objects.

- fourth, how are servers to be implemented: single threaded - risk of distributed deadlock if calls are made to methods of remote objects received as parameters; multi-threaded - the server itself is a concurrent program and needs to rely on some mechanisms for communication and synchronization amongst its processes (this is the source of my remark about it being strange to consider RPC a primitive)

- fifth, how are clients authenticated – a subject for another class

- Example systems than include RPC mechanisms: SunRPC, Java RMI, CORBA, Open Software Foundation/Distributed Computing Environment (OSF/DCE), Microsoft DCOM, SOAP

A note about the history of RPC: at one time there was a fairly fierce battle between RPC believers and RPC non-believers in the distributed systems community. The believers argued that RPC was an desirable basis for all distributed programming. The non-believers said that RPC was unnecessary and that the mechanisms of TCP and UDP were completely adequate for all distributed programming. Both are wrong. The pro RPC argument breaks down fairly quickly when one considers the effect of latency; attempts to overcome the latency problem clutter soon clutter simple, elegant RPC mechanisms with complexity. Also, RPC is not as simple and elegant as one might be led to believe by the simplistic assertion: it's just like local procedure calls: RPCs have different performance and failure attributes than do local procedure calls. These attributes typically do get reflected in the client programs.

On the other hand, research in distributed systems has shown that there are, in fact, well-founded approaches to dealing with the failure and performance issues. TCP and UDP provide raw material for building systems using these approaches, but by building *middleware* above these IP layers we can leverage our understanding of distributed systems issues better than with either RPC or low-level mechanisms.