Lecture 19
Concurrent Programming

5th November 2003

Linda Barrier proofs

So the problem with the Linda barriers is that we really don't have a good way to reason about the behavior: we're making arguments about this happens then that happens, which we've seen is not a viable approach to concurrent programming. What I came up with to tackle this was a translation of the Linda program to a program using conditional critical regions, i.e. await statements.

```
init: b = 1; b2 = n; arrived[1..n] = 0; left[1..n] = 0
co
  /* first process */
  <await b==n>
  b2 = n-1
  <await b2==0>
  b = 1
// /* process i */
  <await b<n; ct=b; b=b+1; arrived[i] += 1>
  <await b2=ct; b2 = b2-1; left[i] += 1>
//...
oc
```

We can then observe that the barrier behavior we are looking for is that captured by a global invariant $\forall i, j \bullet left_i == arrived_j \vee left_i == arrived_j - 1$. We can relate the behavior of the variables b and $b2$ to the that of the auxiliary variables *arrived* and $left$ with the global invariant $(b < n \wedge b = \sum arrived_i - \sum left_i) \vee (b = n \wedge b2 = \sum arrived_i - \sum left_i$. Note that $b < n$ iff we're in the gathering phase of the barrier and $b = n$ when we're in the phase where everyone has arrived and we're waiting for them to leave.

MPI - skip this for lecture; students should read about it as general background

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.

Rendezvous

The rendezvous mechanism was proposed in the late 1970s during the initial competition to develop the Ada programming language. It is reasonable to suppose that the design of rendezvous was influenced by the CSP design, but I'm not sure about that. Like RPC, rendezvous provides a mechanism for a client to invoke an operation in a different context, but unlike RPC the client's operation is serviced by synchronizing with an already-existing process.

Thus, on the client side, rendezvous is very similar to RPC – in both cases the invocation of the remote operation behaves like a procedure call. On the server side, a client's call is handled by an *in* statement (*accept* or *select* statement in Ada). In MPD:

```
in <opname>(<formal parameters>) -> S; ni
```

In Ada:

```
accept <opname> (<formal parameters>) do
   S
end
```

In both MPD and Ada, parameters may be either value parameters or result parameters: value parameters are copied to the in or accept statement at the start of the rendezvous; result parameters are copied to the caller at the end.

To complete the picture we need a way for a server process to offer a choice of operations. In MPD the notation (p. 375) is derived from guarded commands, while Ada uses select (p.400). The MPD implementation of choice also has additional capabilities for prioritizing which branch of the *in* will be chosen and for enabling and disabling branches based on parameter values as well as local state.

Notice that rendezvous does *not* provide selective communication in the generality that we are implementing in project 2: a client can only attempt to perform a single operation at a time.

Example:

A server implementing a bounded buffer:

```
module BoundedBuffer
   op deposit(T), fetch(result T);
body
```

```
   process Buffer{
      T buf[n];
      int front=0, rear=0, count=0;
      while (true) {
         in
            deposit(item) && count<n ->
               buf(rear)=item; rear=(rear+1) mod n; count += 1;
         []
            fetch(item) && count>0 ->
               item=buf(front); front=(front+1) mod n; count=count-1;
         ni
      }
end BoundedBuffer
```

A clients of this bounded buffer invoke operations just as if they were normal procedure calls:

```
process Producer {
   T foo;
   while ... {
      ...
      call BoundedBuffer.deposit(foo)
      ...
   }
}

process Consumer {
   T bar;
   while ... {
      ...
      call BoundedBuffer.fetch(bar)
      ...
   }
}
```

Implementation of rendezvous

Suppose that you were asked to implement a run-time mechanism to support rendezvous. You would have to provide mechanisms to marshall and unmarshall arguments and results, of course. If you had mechanisms to turn argument lists and result lists into message (and vice versa) how would you implement rendezvous?

On the client side, we might tell the compiler writer to translate

```
call M.op(args)
```

to

```
channel M;
...
M.send(marshall(value args))
M.recv(unmarshall(result args))
```

On the server side, let's translate the in statement in

```
module M;
body
process server {
   ...
   in
      op1(args1) and B1 -> S1;
   []
      op2(args2) and B2 -> S2;
   [] ...
      opn(argsn) and Bn -> Sn;
   ni
   ...
}
end M
```

into

```
sl = new SelectionList();
re1 = new RecvEvent(); re2 = new RecvEvent(); ... ren = new RecvEvent()
if B1 then sl.add(re1);
if B2 then sl.add(re2);
...
if Bn then sl.add(ren);
res = sl.select()
if res=re1 then S1(unmarshall(re1.getObject())
elif res=re2 then S2(unmarshall(re2.getObject())
...
elif res=ren then Sn(unmarshall(ren.getObject())
fi
```

Optimizations to the selective communications implementation suggested by this example: we'd like to not allocate a new SelectionList and collection of events for each operation invocation. Suggestions in the project already make the lists and events reusable, but here membership for any particular call of select depends on the boolean guards; perhaps we'd like to allow events to be *enabled* or *disabled* while remaining in a list.

In the MPD version of rendezvous the boolean guards can depend on values in the incoming messages. How could our synchronous communication framework be extended to accomodate that?

Project notes

Can't I just call c.Recv() and c.Send() from the event sync method? yes, but implementing the functions as suggested in the implementation notes will move you farther along toward what is needed for select. Go over how poll and enqueue are just pieces of the existing Send and Recv, pulled out and made accessible. The Send and Recv operations of Channel are no longer needed after this is done and can be removed from the program.