6.826 Principles of Computer Systems **Fall Semester, 1993**

Lecture 18: November 15, 1993

Lecturers: Butler Lampson and Bill Weihl *Scribes: Parry Husbands and Aparna Lakshmiratan*[1]

1 Overview

Today's lecture discusses some issues involved in the implementation of RPC and also the problem of reaching consensus in networks.

2 RPC

The basic idea behind RPC is to emulate the semantics of local procedure calls as closely as possible for reasons of familiarity and distributed system transparency.

The meaning of `y := P(x)` is as follows:

- Find where in the distributed system `P` is available.
- Send the argument(s) `x` and the procedure-id to the remote machine. The argument may have to be encoded (marshalled).
- Get the result, decode it, and pass it to the caller.

2.1 Marshalling

Marshalling is also known as encoding. It's a mechanism for taking the function we want to call as well as its arguments and converting them into a byte stream that can be sent over the network. To do this the encoder must know the types of the function and arguments at run-time. After this the call and the body of the procedure are "hooked up" via a network reference (discussed later).

Some checking needs to be done here. The types must match up and the call and body must somehow agree.

Stubs are used to implement RPCs. The stubs do all the marshalling and dispatch so that a calling thread sees the normal interface of a local procedure call. The stubs may identify the procedure being called using an index into the table of procedures that the remote machine maintains.

The handling of file accesses and nested procedure calls are similar to the handling of object references. In general, what we mean by `P(x)` is actually `x.P`; that is, find the object `x` and invoke its operation `P`. Now, `x` may be a remote object. Finding the remote machine is a part of the linker's job. We may be able to determine this statically, or it may be done dynamically.

[1] These notes are based on those of Srikanth Kannan and Yasuhiro Endo from the Fall of 1992

2.2 Request–Response

RPC imposes the constraint that the caller blocks until the procedure returns. An alternative scheme would be to make the call, return, and then sometime in the future the results will be in the right place. The reason for choosing request-response is that it cuts down on the amount of concurrency in the program. It is widely acknowledged that it is difficult to program and reason about concurrency very well, and so it shouldn't be introduced where it isn't necessary. Non-blocking RPC implicitly creates an extra thread each time a remote procedure call is made. If the caller wants to make an RPC and not block until it returns, she can explicitly create another thread to do the procedure call.

2.3 Network References

Network references give us a way to refer to addresses that may be at remote machines. This is required for several reasons:

- Passing by value is sometimes too expensive. It is much easier to pass a reference to an array of 10,000 elements, for example, than to pass the entire array.
- References make it easier to share code and data. If everything is passed by value, it is very tricky for many different processes to update a variable.
- Not every type of data is mobile. For example, it is not convenient to pick up some code for a procedure and pass it by value. If the procedure is compiled for a Sun4, for example, how can it be passed by value to a VAX? Also the procedure may call many other procedures and these will have to be passed as well.

Network references are therefore a good thing to have. We now discuss how they can be implemented. We can use the technique of hierarchical naming to extend a local address to a network reference. The address `[pid, loc]` in a machine named `n` now gives the network address `(n,[pid,loc])`. Here `pid` is the process identifier and `loc` is a memory location within that process.

Since a network reference can refer to an arbitrary location in any process' address space, the problems of security and robustness come up. It can't always be assumed that the sender of a reference like this knows what he's doing. The way to protect a local machine against bad/unauthorized references is to provide an export table that lists all addresses on that machine that are legal in network references. The sender then gives an index into that table instead of a general address. In the implementation of this we could also add a check field and accept if the request is accompanied by the correct check field. This allows us to re-use table entries.

2.4 Other issues

There are several other implementation issues that arise when dealing with RPC. The reader is encouraged to read Handouts 39 and 40 for more complete presentations.

- Initialization. How do the other machines initially know that references to use? Can a fixed network name service be of use here?

- Storage Management and Garbage collection. We sometimes need to keep working even if some machines fail. How can this be accomplished?
- Binding the remote procedure to the caller. It is obvious that it is impossible to handle this kind of binding in the way that ordinary linkers do. The object oriented approach to RPC that we discussed earlier comes to mind as one desirable option. Consider a case of remote file access below:

  ```
  f := open(...)
  read(f)
  ```

  Note that the conventional static link we are familiar with will not work at all in this case because the location of `read` will not be known until run-time. An object oriented approach seems desirable here: it is much more natural to view `read` as a method of `f`. This approach allows the location of `read` to change based on the location of `f`.

  As attractive as this approach may seem, what we have just done is to trade one problem with another. More precisely, we just traded name to value conversion of `read` to the one of `f`.

  Let us now consider how to look up a file `/rpc/file`. One simple-minded solution is to maintain a conversion table locally. The virtue of this approach is simplicity, but it is not adaptable to changes made to locations of various objects on the network and maintaining such a table could be a tricky and time consuming job.

  Another approach is to establish a name server where the associations between objects and addresses are kept. Each host may export their list of associations to the name server. A typical call to register an association between an object and an address may look as follows:

  ```
  export("/rpc/file", myaddress)
  ```

  Processes can call the name server to obtain the address of the object to determine where to send the RPC packet. A typical call may look as follows:

  ```
  import("/rpc/file")
  ```

  You may notice that these requests are also RPCs. In order to avoid infinite chain of RPCs, we need to make the address of the name server known somehow. This can be done by agreeing upon a universally well known address for the name server(s) or by broadcasting the address of the name server(s) to every node on the network.

3 Consensus

Consensus is used widely in distributed systems. In a distributed system, two or more processes are in consensus if they agree on some value. An application of consensus would be to keep track of which processes are involved in a particular task; in this case, a new consensus must be reached everytime a process fails or starts working.

In general, If you take a deterministic state machine and run it on differnent sites and reach consensus on the values and order of the inputs, they do the same thing.

Today, we will be looking at the specification and implementation of consensus as given in Handout 42. In the first spec on page 2 of Handout 42, we have

`Allow` $\longrightarrow$ $\boxed{\textbf{CONSENSUS with outcome variable}}$ $\longleftarrow$ `Outcome` (returns `nil` or val).

The first spec consists of a sequence of calls to `Allow` and `Outcome`. `Outcome` returns what is in the `outcome` variable, (*i.e.*, val or `nil`). There are two requirements on what `Outcome` can return. The first requirement is agreement which says that all non-`nil` values should agree and the second one is validity which says that `Outcome` only returns allowed values. It is also assumed that if you keep calling `Outcome`, eventually it will return a non-`nil` value.

An example scenario is as follows:

1. `Allow`
2. `Allow`
3. `Allow(d) outcome :=` d
4. `Outcome` can return d or `nil`
5. `Allow`
6. `Allow` *etc*.

In this spec, consensus chooses outcome value at the time of one of the calls to `Allow` but in the implementation outcome doesn't get set at the time of the call to `Allow`.

The next spec in Handout 42 is slightly different from the first one in that `Agree` makes the actual decision about `outcome`. In this case, when the implementation decides it wants to set `outcome`, the chosen value corresponds to an `Agree` action in the specification. Thus, this spec is closer to the implementation and it implements the first spec. In order to prove that the second spec implements the first spec we would need to use prophecy variables.

Note: There is nondeterminism in the specification since at the time when implementation determines outcome, no one may actually know it if the leader fails.

Before moving on to the implementation we need a model for processes that communicate with each other through links. We need to model the failures that can happen.

We have two classes of models:

1. *Synchronous*, where there is a known bound on the amount of time each component takes to make a transition.
2. *Asynchronous*, where component can take an arbitrary amount of time.

In reality, the asynchronous model holds since the network can be congested or a node can be congested. If you send a message to a process, the process might not get cycles for a long time. It is hard to solve an asynchronous system so we add timeouts to the asynchronous system to make it synchronous.

4 Faults

In a system, we can have faulty links in which case no messages get delivered. A fault on a link is equivalent to breaking it. We can also have faulty processes. Process faults can be either stopping faults where a process fails by stopping and doesnt tell you that it has failed or byzantine faults which is equivalent to `HAVOC` where we don't really know anything about what processes do and when they fail. In this discussion, we will focus only on stopping faults. Handout 42 discusses consensus and termination of consensus algorithms in the asynchronous and synchronous models. It has been proved that it is impossible to guarantee the termination of consensus algorithms in all cases. So, the best we can hope for is for the consensus algorithm to terminate with a high probability.

5 Implementing Consensus

5.1 Simple Method

Get a leader, send the leader all the values that we agreed on and the leader decides on the outcome. This is simple but not robust since if the leader fails everyone has to wait.

5.2 Paxos Algorithm

This is an asynchronous algorithm for implementing consensus that gives a high termination probability. The idea is to have one or more leaders which initiate a sequence of ballots to which the voters respond. Ballots are identified by a unique number. The ballot contains the ballot ID and a value. The leader does the following in each ballot:

1. It tells voters it is beginning a new ballot, and queries them for their voting histories. Voters must now decide to vote "no" on any previous ballots for which they have not yet voted.

2. When a majority of the voters have responded to the query, the leader chooses the value to be used for the ballot. If any voter has previously voted "yes" for a value d, the most recent such value is chosen. Otherwise, the leader choses its own value for the ballot.

3. The leader polls the voters to get a majority to accept the value chosen.

4. If the poll is successful, the leader broadcasts the value.

Step 2 above guarantees validity, since the value chosen for the ballot is selected from querying the voters about voting history. This step also makes sure that any consensus reached will not be lost due to a failure, since the leader will definitely know of any consensus reached in the past when he waits for query responses from a majority of voters. If the leader gets even one "yes" vote among the majority who responded in a particular query for a value d, the leader chooses d for the poll using the logic that if there was a majority that voted "yes", d must have been the value.

The voter votes "yes" unless he has already voted "no". The ballots are totally ordered. A ballot is "dead" if a majority voted "no" on the value of that ballot. In this scheme, if a leader fails between getting messages back and broadcasting the outcome then we can elect a new leader. The new leader should start a new ballot and choose the same value to vote on after the query stage if the outcome had already been decided.

As an example let us consider the following scenario with 2 leaders and 3 voters.

L_1	V_1	V_2	V_3	L_2
query (ballot 1)				
	ret $\{\}$	ret $\{\}$	ret $\{\}$	
choose d				
poll (1, d)				
	yes(1, d)	yes(1, d)		
crash				

At this point majority have answered "yes" and outcome is decided. Any later ballot chooses the same value. Now L_1 crashes before broadcasting outcome.

				query (ballot 3)
	ret $\{$yes(1,d)$\}$		ret $\{$no(1)$\}$	
				choose d
				poll(3,d)

L_2 chooses the value d for ballot 3 since from the query L_2 decided that if there was a majority that voted "yes", d was the value. Thus, L_2 chooses d even though it saw only one "yes".

The working of the algorithm and more examples will be discussed next time.