6.826 Principles of Computer Systems	**Fall Semester, 1992**

Lecture 18: November 18, 1992

Lecturer: Butler Lampson | *Scribe: Yasuhiro Endo*

1 Remote Procedure Calls (Continued)

We have not considered how to bind a 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. Object oriented approach to RPC that we have discussed in Lecture 17 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. 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 to where the associations between object and address 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 name server known somehow. This can be done by agreeing upon universally well know address for the name server(s) or by broadcasting the address of the name server(s) to every node on the network.

2 Failures

Until recently, people thought it was valid to assume that faulty processes just stop making transitions. These failures are called *stopping faults*. As computer systems start to take on roles that require higher degree of fault tolerances (e.g. aircraft control), old conceptions about faulty processes were re-considered. As a result, a new classification of faults called *Byzantine faults* was introduced. A process with Byzantine fault makes arbitrary transitions instead of halting. Faulty processes of this kind have much more serious consequences, for it is possible for faulty processes to "confuse" normal processes.

For the following discussions, we will concern ourselves with only stopping faults.

3 Consensus

One way to achieve reliability on a distributed system is to have many nodes do the same thing and take a vote on the result. The concept itself is simple, but having all the nodes on a distributed system is rather complicated task. There are two conditions that need to be satisfied in order to do have many nodes do the same. First, all the node involved in this business must be exactly equal deterministic state machines. Second, it must be made sure all the state machines are fed the same sequence of input.

We might accomplish sequencing of requests by pairing a request with a unique number that is constantly increasing. This will work for a single client case. However, since it is highly likely that this replicated service have multiple clients, this simple minded solution will not suffice in most of real world cases. Some kind of time stamps are used instead of a simple number to introduce orders in requests filed by multiple clients.

We now have a mechanism to maintain an order in requests, but we still do not have a mechanism to have every node agree on what to perform next. One common way to achieve this is called primary copy scheme. This scheme relies on a "boss" server which decides what to perform next and lets the other know of its decision. Because there is only one node that is capable of making a decision as to what to do next, consensus is guaranteed in this scheme. As you may notice, this scheme is not resilient to the crash of the central node, for this scheme relies heavily on the services performed by the central node.

It has been proven that it is impossible to guarantee the termination of consensus algorithm under all the situations. Circumstances under which we can guarantee the termination of the algorithm are discussed in detail in handout 43. Because of this known limitation of consensus algorithm, the most we can hope for is an algorithm that terminates with high probability.

4 Paxos Algorithm

This algorithm is one of the algorithms that give us higher termination probability. Basic idea of this algorithm is to have participants respond to a sequence of ballots initiated by leader(s). Ballots are identified by a number that is unique. Identifiers are assigned in increasing order. A ballot is a pair that contains the unique ballot id and a value.

Leader host(s) go through following steps to reach consensus:

1. First thing to do is to start a new ballot and send *collect-past-information* messages to gather voters' voting history.

2. When responses from a majority of the voters are received, the value d to be used for the ballot is computed. This value is obtained from the highest ballot number to which any voter voted "yes". If there are no "yes" responses in the histories received, as would be the case with the very first ballot, then the leader's own preference is used.

3. Send *collect-votes* messages, asking participants to vote for the ballot.

4. If a majority votes "yes" to the ballot, accept the value of the ballot as the consensus value.

5. Broadcast the consensus value.

Note that step 2 above is the one that guarantees validity because the value chosen for the ballot is selected from some voter's voting history or from the leader's preference. If selected from a voter's history, we can apply the same argument for the corresponding ballot; thus the value must have originated as some leader's preference. The same step is also important in guaranteeing that any consensus reached will not be lost. By waiting for responses from a majority of voters, the leader is sure to have received information about any consensus reached in the past. This is true because consensus is reached only when a majority votes yes, and this majority must intersect with the one the leader heard from.

Each participating voter acts based on the rules described below:

1. When it receives a collect-past-information message, it responds by sending back a packet with the number and the value of the ballot for which it last voted yes, if any. Also, if it has not yet voted for a smaller ballot, it decides to vote "no" for it (the smaller ballot).

2. When it receives a collect-vote message, it votes yes unless it had already decided "no" for this ballot in response to some collect-past-information message for a higher ballot (see step 1).

Examples on the working of the algorithm will be studied in the next lecture.