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

Lecture 17: Nov 16, 1992

Lecturer: Butler Lampson — *Scribe: Srikanth Kannan*

1 Overview

Topics for today:

- Implementations of Transport layer
  - The LSW Protocol
  - Three Way Handshake
- The Session Layer (RPC)

Handout 38 accompanies this lecture.

2 The LSW Protocol

Recall that the sender and receiver used identifiers chosen from good sets to achieve at most once semantics. The condition on the good sets is

$$g_s \subseteq g_r$$

The LSW protocol uses *time-stamps* as the identifiers. The sender marks the messages with it's current time. The receiver maintains a window of good identifiers. This window is bounded by *l*, the lower bound and *u*, the upper bound. The values of *l* and *u* depend on ϵ, the clock skew at any node, and on δ, the end-to-end network latency including the time required for retransmission to guarantee delivery with a very high probability. The upper bound is kept higher than $t + 2 * \epsilon$, where *t* is the current time, so that zero-latency messages from a node with a fast clock get accepted. On the other hand, the lower bound can be raised to $t - \delta - 2 * \epsilon$, because messages timestamped below this must be delayed duplicates. Also, we can update the *l* to t_p if a packet with time stamp t_p arrives. So, if t_{last} is the time stamp of the last packet received,

$$l = max\{t_{last}, t - \delta - 2 * \epsilon\}$$

This means that we don't have to maintain the connection information for idle senders. If a sender has not sent packets for quite a while, the t_{last} for this connection is going to fall below $l = t - \delta - 2 * \epsilon$. Since we maintain *l* for active connections anyway, we can remove the idle senders from the connection table. A new packet received from this sender will be accepted if and only if it is timestamped greater than $t - \delta - 2 * \epsilon$. This guarantees that duplicate messages will not be accepted, without having to maintain information about the last connection. It does mean that a message delayed by more than the bound on latency might be mistaken for a delayed duplicate and be dropped by the receiver.

There is one problem in forgetting t_{last}, however. The receiver must send positive acks for duplicate messages with this timestamp. On the other hand, a message older than t_{last} has to be acknowledged negatively because the receiver cannot be sure that it was delivered. The receiver must therefore maintain t_{last} until it knows that the sender has either received the ack or has timed out. This can be achieved by bounding the time for the ack to be delivered to the sender.

3 Three Way Handshake Protocol

Again, we require

$$g_s \subseteq g_r$$

In this protocol, the good identifiers are found by a handshake between the sender and the receiver. The handshake works by passing 5 packets between the sender and the receiver. The sender gets a good identifier from the receiver and sends packet(s) on this identifier (figures on page 7 of Handout 38).

The protocol delivers messages at most once and passes messages correctly even when the sender or the receiver crashes.

One situation we need to consider is when a sender sees a duplicate good-identifier packet `(accepting, ri, si)` that the receiver sent. The sender should be able to tell if this packet was sent in response to it's recent request or if it is just an old packet popping up now. To be able to do this, the sender should not reuse the `si` (or, only reuse it with care). Then, when a `(accepting, ri, si)` packet comes, the sender can check the `si` to ascertain that the packet is really for it's most recent request. The way that the sender can avoid using the `si` across crashes is by writing the `si`s onto stable storage. But such operations are costly and we would like to minimize them. What the sender can do is to write say 1000 `si`s onto disk, so it can use these 1000 before another disk write. ln the rare event of a crash, it will lose some good `si`s, but that is no big deal.

4 Session Layer

Now we move to session layer. Since nobody really knows what this layer is for, we will deal with RPCs. The basic idea behind the 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.

What if the arguments contain memory references to other objects? One solution is to copy the contents of the referenced objects as well, and send the entire structure to the callee. This gets too cumbersome for a large graph of connected objects. A common technique is to send *net-references* for the referenced objects. If the callee wants to access such an object, it can use its net-reference to make a call-back and get the contents of the object. The net-reference contains enough information that allows the callee to make such a call-back.

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.