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

Lecture 16: November 8, 1992

Lecturer: Butler Lampson *Scribe: Tina Kapur and Oded Maron*[1]

1 Overview

Today's lecture covers the following topics:

- Switching functions and routing methods
- Learning network topology
- Reliable message passing

2 Switching functions

Last time, we described the topology of a network as:

```
T : L -> SS
```

i.e., a function from links to source-sink pairs of nodes. Topology, however, does not answer the question of what to do with a routed package. With broadcasting, the answer is simple: the router doesn't do much other than broadcast messages it has received. In general switched networks, we need more sophisticated routing techniques. We'll use the concept of a *switching function* to describe the routing used:

`SW: R -> A -> L` *or* `SW: R -> A -> SET L`

This means, that given a router, the switching function will map an address to a link (or set of links).

We would like the switching function to give us an optimal (least cost) path, within some tolerance. A simple cost function could be

```
Cost: P -> Int
```

where the cost is the length of the path.

We note that in order to find the least cost switching function, we need to know the topology of the network. Also, sometimes we will need to rename addresses as they go through routers in order to simplify our life. We then create a mapping function, which given a router and an address, returns a new address.

```
Map: R -> A -> A
```

[1] These notes are based on those of David Maw of November 9, 1992

3 Routing methods

3.1 Source Routing

We can make the address specify which links to use in order to deliver the message. This way, the switching function is trivial since all it needs to do is look at the head of the address. The mapping function is also straightforward—simply strip off the head.

```
SW = a.head
Map = a.tail
```

There area few problems with this method: the sender must know the entire network topology; it is hard to deal with congestion and alternate routings; the address is variable length; if the topology is dynamic, this method fails.

3.2 Distributed source routing

In order to deal with some of the fault tolerance concerns that are not addressed by source routing, the friendly people at AT&T came up with the idea of distributing the routing information through the network. We place a "virtual circuit" at each router. We rename the address at every router by using a table lookup. The router only needs to know about the types of incoming addresses, and that limits the size of the table. The host only needs to know about the local topology.

However, there are still problems. Setting up the lookup tables is expensive for the router, especially for large networks. This method is used mostly by telephone companies, but it is also the basis for Asynchronous Transfer Mode.

3.3 Global addressing

In each of the above methods, addresses are relative. In other words, they have different meanings at different places in the network. In this new scheme, we have global addressing. There can be two types of global addressing schemes: flat and hierarchical. (Can you guess which is better?) Flat global addressing leads to huge lookup tables and used mostly in small LANs.

Hierarchical addressing takes advantage of a tree-like structure which is imposed on the network. Each node knows how to route messages to its children, its parent, and its siblings. Therefore, each node needs a very small amount of routing information (specifically, twice the fanout of the tree). If the address is not a child, parent, or sibling, the node sends it up to its parent, and so on recursively.

4 Learning topologies

How do we set up these routing tables? If we know the topology, then we simply use a shortest path algorithm (*e.g.*, [Dijkstra, Tarjan, Olajoun 89]). However, what if the topology is not known and needs to be learned? Following are three methods to learn network topology.

4.1 Epoch

We assume that the topology doesn't change within an epoch. We run a spanning tree algorithm to pick a root and impose a partial order on the nodes at the beginning of each epoch. Each node sends its neighbor information to the root. The root then processes the information and sends the relevant topological information to each node.

Problems are: if the network is big, then this will take a lot of time. Also, the length of an epoch must be smaller than the rate of change of the topology.

4.2 Incremental flooding

As opposed to the epoch method, this is a local, incomplete method. Each router broadcasts information about itself to its neighboring routers. The topological information is collected incrementally and is used to find shortest paths. However, all information need not agree at all times, and the routing scheme must be able to handle these inconsistencies. If network configuration changes in an "unfortunate" way, the network can end up in a temporary or permanent loop. The general rule to deal with this problem is to discard packets that have been in the network for too long.

4.3 Bridges

This is a flat learning scheme. Each router can have a *partial* switching function. In other words, it knows how to deliver to certain addresses, but not to others. If it receives an unknown address, it broadcasts it in the hope that other routers will know how to deliver the message. The router learns by adding information to its switching table. When a node sends a message, it appends its own address to it so that path information can be sent back to it. We use a spanning tree to limit cycles.

5 Network Issues

5.1 Switching and Addressing

We've talked about this above.

5.2 Arbitration

- Flow Control—the receiver needs to tell the sender when to stop. There are two types of methods for doing this: explicit scheduling (start-stop and credit methods) and exponential backoff.

  A common explicit scheduling strategy is called flow-control or backpressure. One scheme is to grant each node some credits such that it can send as much as it is granted. When a node runs out of credits, it must ask for more credits for sending. Alternatively, a node may start to send any time and send until a stop is signaled on a back channel. However, it is quite tricky to decide when to send a stop and how much buffer each node should have to queue up the data that arrives subsequently. In general, at the time of sending a start signal, the receiving node must have enough buffer space to hold at least two round-trip time of full-rate sending of data. One for stop to take effect and another for smooth flow in the network. On the other hand, the credit-based method is immune to buffer shortage problem because credits are granted depending upon the free buffers available. However, a credit-based scheme can get out-of-synchronization. The credit messages are not idempotent and thus cannot be simply resent when they are lost. A more complicated scheme is necessary for recovering the credit messages in both directions. Both schemes of explicit scheduling strategy take up a considerable about of resource.

Another simple approach is a backoff scheme which allows dropping of data when buffer is overloaded. There are two strategies: One is to estimate the load of a switched network by collecting a lot of statistics on the usage of the network, and decide a maximum capacity of the network. The telephone companies use this method and simply decline service when infrequent occasions of bursty overload occur. However, it is much harder to collect statistics on a data-communication network which has no central switch and has much more bursty traffic than the telephone switch network. A more systematic backoff scheme is desirable to estimate how much load there is in a data-communication network and make sure each node gets its fair share. The backoff method used for a long-haul network like the Internet is very similar to that used for the Ethernet, except that each node should be cautious when it starts to send data without a good estimate of the load. Since a node in the Internet does not always monitor the load of the network and it will take a long time after a node starts to send data in order to get any feedback about the load of the network with the backoff scheme, a node should have a slow start in sending data. The slow start algorithm prevents a node from jamming the network at the start. A sending node should wait for feedback about the load of the network and gradually increase the rate of data sending to match up with the estimate of the load. All in all, the backoff method requires all nodes in the network to follow the same algorithms so that no single node can have an unfair advantage over others.

- Discard overload—consistent with spec as seen below.

5.3 Congestion

The hosts need to worry about preventing overflow. One option is to use backoff, but to use this method the hosts need to know about collisions. TCP provides this information (through acknowledgments) but there might be a long lag from the time you send a message until you discover its loss. Therefore, we need to increase/decrease load slowly. This is known as *slow start*.

5.4 Availability

- Use multiple paths
- Propagate topology

6 Reliable Messages

Let us present a simplified spec, which only worries about one message, and its loss or duplication.

```
q: SEQ M      % or actually a single send-receive pair

Put(m) = q := q++m;

Get() -> M = VAR m | q # {} => m:=q.head; q:=q.tail; RET m
```

How do we implement the above spec on top of the unreliable network discussed so far? In order to deal with failures, we'll need to change stable storage every time we do a `Get`.

```
P = [i,m]  % a packet consisting of unique id and a message
ch: SET P  % a channel which represents the state of the world
```

We need an infinite source of unique ids. We have this source for both the sender and the receiver. The sender needs it to generate fresh ids, and the receiver needs it to check if it has gotten a duplicate message. We keep all of this information in stable storage:

```
   new_s, new_r : SET I    % set of unique ids
   msg: M
   last: I                 % last id used

% Implementation

Put(m) = msg = nil =>
  VAR i | i IN new_s =>
            msg := m; last := i;
            new_s := new_s - i

Send() = msg # nil => ch.send(i,m);

RecAck() = VAR i:=ch.receive; i = last => msg := nil

Get() -> M =
   DO VAR (i,m) := ch.rec() |
        i IN new_r => new_r:= new_r - i;
                      ch.send(i);           % acknowledgment
                      RET m
                  [*] ch.send(i)            % ack in any case
   OD
```

The abstraction function is:

```
q = {msg} if last IN new_r
  = {}    otherwise
```

It should be obvious how to extend this implementation for more messages (ha!).

This implementation is not usually used in communication, but in queueing since you need frequent accesses to stable storage (which is slow) and it doesn't take into account losing messages. A more realistic implementation is given below.

We create a volatile version of `new_s`, called `vnew`. Also, `msg` and `last` are now volatile. We also need to add the possibility of a crash to the spec, and also modify the implementation for `Put`:

```
Crash() = q:= {} [] SKIP  % depending on whether msg was delivered
                          % before or after the crash
Put(m) =
  msg = nil =>
     vnew = {} => fill it up from new_s
         VAR i | i IN vnew =>
                     msg := m; last := i;
                     vnew := vnew - i
```

The implementation for crash is now:

```
Crash() = vnew:={}; msg:=nil;
```

Is the abstraction function still correct? The representation invariant used to be:

```
(i,m) IN ch /\ i IN new_r => m = msg
```

However, if the sender crashes, `msg` becomes `nil` and the invariant is not maintained. Tune in next time for the exciting conclusion when we find out what the correct abstraction function is.