6.826 Principles of Computer Systems	**Fall Semester, 1992**
Lecture 16: November 9, 1992	
Lecturer: Butler Lampson	*Scribe: David Maw*

1 Overview

Today's lecture continues discussion on the ISO Reference Model. Topics covered include:

- Congestion Control on Network Layer
- A Specification of Transport Layer
- Implementations of Transport Layer

Note: Handout 38 "Reliable At Most Once Messages" accompanies this lecture with specification and implementations of the Transport Layer.

2 Congestion Control on Network Layer

There are two types of networks discussed in the last lecture, broadcast and switched. Once the arbitration issue of a broadcast network is solved, there is no problem of congestion. For the switched network, a given sender does not have any information about what the other senders are doing; therefore, it is quite possible that it sends more than a network can absorb. There are basically two strategies to control congestion on a switched network, *explicit scheduling* and *backoff*. Generally speaking, explicit scheduling has one node gathering requests for a resource and explicitly handing out grants for the resource. On the other hand, the backoff strategy allows each node to send as much as it wants and somehow backoff when it detects that the network is dropping its messages.

2.1 Explicit Scheduling: Credit-based and Start/Stop Schemes

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.

2.2 Backoff

Another simply approach is backoff 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, the statistics on a data-communication network is much harder to collect 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 have each node get 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.

3 Transport Layer

The layer above the Network Layer of the ISO model is the Transport Layer which provides reliable message delivery. Reliable delivery means that all messages sent are received in order and are acknowledged. However, in fact, the Transport Layer cannot handle some types of end-node failures which can result in loss of messages. [Refer to Handout 38 page 1]

3.1 Specification

Page 2 of Handout 38 describes the specifications of a faulty Network Layer network and Page 3 of Handout 38 describes the reliable at-most-once message delivery service at the Transport Layer. The specifications state that there is a FIFO queue with two basic operations: Put and Get. An additional operation, GetAck, is required to convey the acknowledgments. In the spec given below, failures are handled in a somewhat different way than that in Handout 38. Two more variables, rec_s and rec_r, are introduced for recovery, and separate Recovery actions and Lose operation have been added.

APROC CrashS = $\langle\langle$ rec_s := true $\rangle\rangle$
APROC CrashR = $\langle\langle$ rec_r := true $\rangle\rangle$
APROC RecoverS = $\langle\langle$ rec_s := false $\rangle\rangle$
APROC RecoverR = $\langle\langle$ rec_r := false $\rangle\rangle$
APROC Lose =
    $\langle\langle$ $rec_s \vee rec_r \Rightarrow$
        VAR q1, q2, m |
            q = q1 + m + q2 $\Rightarrow$ q := q1 +q2;
                q2 = {} $\Rightarrow$ ack := false
        [] BEGIN ack := false [$*$] SKIP END $\rangle\rangle$

3.2 Implementation

In a general implementation model for the specification described above, each message is tagged with an identifier. Both the sender and the receiver have sets of identifiers. The receiver has a set of good identifiers, g_r, which are identifiers the receiver knows have not been attached to any message received. We assume that the identifiers are ordered and there is a property that all identifiers in g_r are bigger than the identifiers of messages received. Upon receiving a message, the receiver checks to see if the identifier of the message is in the g_r, if so, the receiver will take it, otherwise, the message is dropped as duplicate. The sender also chooses identifier from its own set of good identifiers, g_s, which satisfies the property:

$$g_s \subseteq g_r$$

This is so because if the sender picks an identifier that the receiver does not think is good and transmits it, the receiver will drop the message. That will violate the protocol, which says all the messages are supposed to be accepted unless there is a crash.

In the simple case of a one-message buffer, the basic notion of the protocol is that when the sender is idle, it send out a message, *m*, with an identifier, *last*, picked from g_s. The sender keeps the current message (last message sent) in a variable, *c* = (*last*,*m*) and retransmits it until a positive acknowledgment is received. Since *last* is picked from g_s, a subset of g_r, the receiver is ready to take message *m* with identifier *last*. The identifier *last* is removed from the g_s when it is picked by the sender and is subsequently removed from g_r when the receiver gets it.

The receiver accepts a message if it is tagged with an identifier that is in its good set, and then removes the identifier from the good set. If the identifier is not in its good set, but was the last message accepted by the receiver, it sends back a positive ack. Otherwise, a *negative* ack is sent. Note that there are several reasons why such a case may arise. First, it might be a delayed duplicate of an old message; since it is not the current message being sent by the sender, the sender will ignore the negative ack. Second, the receiver might have crashed before accepting any copy of the message, and as a result of the crash, the identifier got removed from its good set. Subsequently, if it recovered and received a message with that identifier, it would reject the message and send back a negative ack, which will inform the sender of the loss, as required by the spec. Third, the receiver might have accepted the first copy of the message, removed the identifier from its good set, and sent a positive ack that did not reach the sender. If it then crashed, and subsequently received a retransmitted copy of the message, it would send a negative ack because it forgot the identifier of the last message it accepted. The sender would see the negative ack although the message did get delivered, but this is ok because it is allowed by the spec.

We now try to develop an abstraction function from the state of the implementation to that of the spec. A first cut might be:

$$\begin{aligned} q &= \{c\}, \text{if } c \neq nil \wedge last_s \in g_r \\ &= \{\}, \text{otherwise} \end{aligned}$$

this says that a message gets into the abstract state q as soon as c is set, and is removed from there when it is accepted by the receiver (indicated by the removal of the identifier from g_r).

This is incomplete because if the sender crashes, c is nullified, but the message may be buffered in the network, and may actually get delivered to the receiver. To account for such messages, q is augmented as follows:

$$\begin{aligned} q = \; & \{(i,m) | (i,m) \in s_r \wedge i \neq last_s \wedge i \in g_r\}.sorted \\ & + \{c\}, \text{if } c \neq nil \wedge last_s \in g_r \end{aligned}$$

If we try to prove that the implementation satisfies the spec using the above abstraction function, we hit a serious snag illustrated below:

1. sender sends message $(1, m1)$ into the network.
2. sender crashes.
3. sender recovers.
4. sender sends message $(2, m2)$ into the network.
5. the network loses $(1, m1)$.

According to the abstraction function, q has the following values at different stages:

1. $\{m1\}$
2. $\{m1\}$
3. $\{m1\}$
4. $\{m1, m2\}$
5. $\{m2\}$

The problem is that although the spec allows loss of messages when there is a crash, it restricts the loss to occur between the crash and recovery. But in the above scenario, the sender has recovered in stage 3 (it even sent a new message $m2$), and the loss seems to have occurred *after* all that. Thus, the implementation is making the decision to lose messages later than when the spec allows it.

As far as externally visible behavior is concerned, the implementation does satisfy the given spec. However, given the tools introduced so far (including history variables, etc.), it is impossible to come up with an abstraction function that will prove it so. We need another fix, called *prophecy variable*, which is in some sense the dual of a history variable. But instead of using prophecy variables, we shall simply change the spec so that the implementation can be proved correct.

The idea is that instead of losing messages on a crash, the spec would simply *mark* them for possible later loss. Each message in q has a mark bit associated with it:

q = SEQ [m, mark]
APROC CrashS = $\langle\langle$ rec_s := true $\rangle\rangle$
APROC CrashR = $\langle\langle$ rec_r := true $\rangle\rangle$
APROC RecoverS = $\langle\langle$ rec_s := false $\rangle\rangle$
APROC RecoverR = $\langle\langle$ rec_r := false $\rangle\rangle$
APROC Mark =
    $\langle\langle$ $rec_s \vee rec_r \Rightarrow$
        $\%$ mark messages in q
APROC Drop =
    $\%$ delete a marked message from q