5.3.1 Bulk Transfer (BLAST)

The first problem we are going to tackle is how to turn an underlying network that delivers messages of some small size (say, 1 KB) into a service that delivers messages of a much larger size (say, 32 KB). While 32 KB does not qualify as "arbitrarily large," it is large enough to be of practical use for many applications, including most distributed file systems. Ultimately, a stream-based protocol like TCP (see Section 5.2) will be needed to support an arbitrarily large message, since any message-oriented protocol will necessarily have some upper limit to the size of the message it can handle, and you can always imagine needing to transmit a message that is larger than this limit.

We have already examined the basic technique that is used to transmit a large message over a network that can accommodate only smaller messages—fragmentation and reassembly. We now describe the BLAST protocol, which uses this technique. One of the unique properties of BLAST is how hard it tries to deliver all the fragments of a message. Unlike the AAL segmentation/reassembly mechanism used with ATM (see Section 3.3) or the IP fragmentation/reassembly mechanism (see Section 4.1), BLAST attempts to recover from dropped fragments by retransmitting them. However, BLAST does not go so far as to *guarantee* message delivery. The significance of this design choice will become clear later in this section.

What Layer Is RPC?

Once again, the "What layer is this?" issue raises its ugly head. To many people, especially those who adhere to the Internet architecture, RPC is implemented on top of a transport protocol (usually UDP) and so cannot itself (by definition) be a transport protocol. It is equally valid, however, to argue that the Internet should have an RPC protocol, since it offers a process-to-process service that is fundamentally different from that offered by TCP and UDP. The usual response to such a suggestion, however, is that the Internet architecture does not prohibit network designers from implementing their own RPC protocol on top of UDP. (In general, UDP is viewed as the Internet architecture's "escape hatch," since effectively it just adds a layer of demultiplexing to IP.) Whichever side of the issue of whether the Internet should have an official RPC protocol you support, the important point is that the way you implement RPC in the Internet architecture says nothing about whether RPC should be

BLAST Algorithm

The basic idea of BLAST is for the sender to break a large message passed to it by some high-level protocol into a set of smaller fragments, and then for it to transmit considered a transport protocol or not.

Interestingly, there are other people who believe that RPC is the most interesting protocol in the world and that TCP/IP is just what you do when you want to go "off site." This is the predominant view of the operating systems community, which has built countless OS kernels for distributed systems that contain exactly one protocol—you guessed it, RPC—running on top of a network device driver.

The water gets even muddier when you implement RPC as a combination of three different microprotocols, as is the case in this section. In such a situation, which of the three is the "transport" protocol? Our answer to this question is that any protocol that offers process-to-process service, as opposed to node-to-node or host-to-host service, qualifies as a transport protocol. Thus, RPC is a transport protocol and, in fact, can be implemented from a combination of microprotocols that are themselves valid transport protocols.

these fragments back-to-back over the network. Hence the name BLAST—the protocol does not wait for any of the fragments to be acknowledged before sending the next. The receiver then sends a *selective retransmission request* (SRR) back to the sender, indicating which fragments arrived and which did not. (The SRR message is sometimes called a *partial* or *selective* acknowledgment.) Finally, the sender retransmits the missing fragments. In the case in which all the fragments have arrived, the SRR serves to fully acknowledge the message. Figure 5.13 gives a representative timeline for the BLAST protocol.

We now consider the send and receive sides of BLAST in more detail. On the sending side, after fragmenting the message and transmitting each of the fragments, the sender sets a timer called DONE. Whenever an SRR arrives, the sender retransmits the requested fragments and resets timer DONE. Should the SRR indicate that all the fragments have arrived, the sender frees its copy of the message and cancels timer DONE. If timer DONE ever expires, the sender frees its copy of the message; that is, it gives up.

On the receiving side, whenever the first fragment of a message arrives, the receiver initializes a data structure to hold the individual fragments as they arrive and sets a timer LAST_FRAG. This timer counts the time that has elapsed since the last fragment arrived. Each time a fragment for that message arrives, the receiver adds it to this data structure, and should all the fragments then be present, it reassembles them into a complete message and passes this message up to the higher-level protocol. There are four exceptional conditions, however, that the receiver watches for:

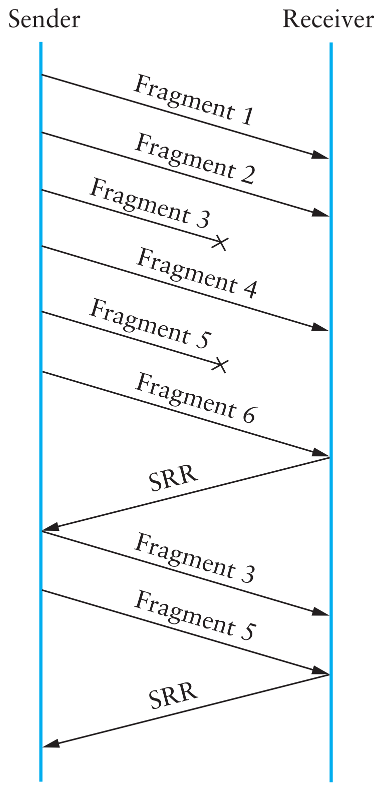

Figure 5.13 Representative timeline for BLAST.

- If the last fragment arrives (the last fragment is specially marked) but the message is not complete, then the receiver determines which fragments are missing and sends an SRR to the sender. It also sets a timer called RETRY.
- If timer LAST_FRAG expires, then the receiver determines which fragments are missing and sends an SRR to the sender. It also sets timer RETRY.
- If timer RETRY expires for the first or second time, then the receiver determines which fragments are still missing and retransmits an SRR message.
- If timer RETRY expires for the third time, then the receiver frees the fragments that have arrived and cancels timer LAST_FRAG; that is, it gives up.

There are three aspects of BLAST worth noting. First, two different events trigger the initial transmission of an SRR: the arrival of the last fragment and the firing of the LAST_FRAG timer. In the case of the former, because the network may reorder packets, the arrival of the last fragment does not necessarily imply that an earlier fragment is missing (it may just be late in arriving), but since this is the most likely explanation, BLAST aggressively sends an SRR message. In the latter case, we deduce that the last fragment was either lost or seriously delayed.

Second, the performance of BLAST does not critically depend on how carefully the timers are set. Timer DONE is used only to decide that it is time to give up and delete the message that is currently being worked on. This timer can be set to a fairly large value since its only purpose is to reclaim storage. Timer RETRY is only used to retransmit an SRR message. Any time the situation is so bad that a protocol is reexecuting a failure recovery process, performance is the last thing on its mind. Finally, timer LAST_FRAG has the potential to influence performance—it sometimes triggers the sending by the receiver of an SRR message—but this is an unlikely event: It only happens when the last fragment of the message happens to get dropped in the network.

Third, while BLAST is persistent in asking for and retransmitting missing fragments, it does not guarantee that the complete message will be delivered. To understand this, suppose that a message consists of only one or two fragments and that these fragments are lost. The receiver will never send an SRR, and the sender's DONE timer will eventually expire, causing the sender to release the message. To guarantee delivery, BLAST would need for the sender to time out if it does not receive an SRR and then retransmit the last set of fragments it had transmitted. While BLAST certainly could have been designed to do this, we chose not to because the purpose of BLAST is to deliver large messages, not to guarantee message delivery. Other protocols can be configured on top of BLAST to guarantee message delivery. You might wonder why we put any retransmission capability at all into BLAST if we need to put a guaranteed delivery mechanism above it anyway. The reason is that we'd prefer to retransmit only those fragments that were lost rather than having to retransmit the entire larger message whenever one fragment is lost. So we get the guarantees from the higher-level protocol but some improved efficiency by retransmitting fragments in BLAST.

BLAST Message Format

The BLAST header has to convey several pieces of information. First, it must contain some sort of message identifier so that all the fragments that belong to the same message can be identified. Second, there must be a way to identify where in the original message the individual fragments fit, and likewise, an SRR must be able to indicate which fragments have arrived and which are missing. Third, there must be a way to distinguish the last fragment, so that the receiver knows when it is time to check to see if all the fragments have arrived. Finally, it must be possible to distinguish a data

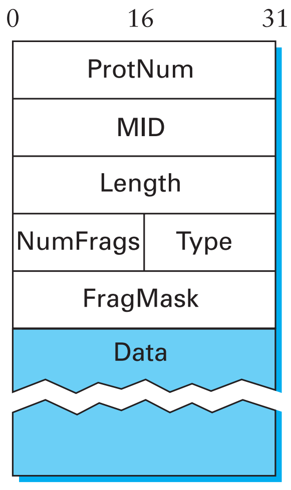

Figure 5.14 Format for BLAST message header.

message from an SRR message. Some of these items are encoded in a header field in an obvious way, but others can be done in a variety of different ways. Figure 5.14 gives the header format used by BLAST. The following discussion explains the various fields and considers alternative designs.

The MID field uniquely identifies this message. All fragments that belong to the same message have the same value in their MID field. The only question is how many bits are needed for this field. This is similar to the question of how many bits are needed in the SequenceNum field for TCP. The central issue in deciding how many bits to use in the MID field has to do with how long it will take before this field wraps around and the protocol starts using message ids over again. If this happens too soon—that is, the MID field is only a few bits long—then it is possible for the protocol to become confused by a message that was delayed in the network, so that an old incarnation of some message id is mistaken for a new incarnation of that same id. So, how many bits are enough to ensure that the amount of time it takes for the MID field to wrap around is longer than the amount of time a message can potentially be delayed in the network?

In the worst-case scenario, each BLAST message contains a single fragment that is 1 byte long, which means that BLAST might need to generate a new MID for every byte it sends. On a 10-Mbps Ethernet, this would mean generating a new MID roughly once every microsecond, while on a 1.2-Gbps STS-24 link, a new MID would be required once every 7 nanoseconds. Of course, this is a ridiculously conservative calculation—the overhead involved in preparing a message is going to be more than a microsecond. Thus, suppose a new MID is potentially needed once every microsecond, and a message may be delayed in the network for up to 60 seconds (our standard worst-case assumption for the Internet); then we need to ensure that there are more than 60 million MID values. While a 26-bit field would be sufficient ($2^{26} = 67{,}108{,}864$), it is easier to deal with header fields that are even multiples of a byte, so we will settle on a 32-bit MID field.

This conservative (you could say paranoid) analysis of the MID field illustrates an important point. When designing a transport protocol, it is tempting to take shortcuts, since not all networks suffer from all the problems listed in the problem statement at the beginning of this chapter. For example, messages do not get stuck in an Ethernet for 60 seconds, and similarly, it is physically impossible to reorder messages on an Ethernet segment. The problem with this way of thinking, however, is that if you want the transport protocol to work over any kind of network, then you have to *design for the worst case*. This is because the real danger is that as soon as you assume that an Ethernet does not reorder packets, someone will come along and put a bridge or a router in the middle of it.

Let's move on to the other fields in the BLAST header. The Type field indicates whether this is a DATA message or an SRR message. Notice that while we certainly don't need 16 bits to represent these two types, as a general rule we like to keep the header fields aligned on 32-bit (word) boundaries, so as to improve processing efficiency. The ProtNum field identifies the high-level protocol that is configured on top of BLAST; incoming messages are demultiplexed to this protocol. The Length field indicates how many bytes of data are in *this* fragment; it has nothing to do with the length of the entire message. The NumFrags field indicates how many fragments are in this message. This field is used to determine when the last fragment has been received. An alternative is to include a flag that is only set for the last fragment.

Finally, the FragMask field is used to distinguish among fragments. It is a 32-bit field that is used as a bit mask. For messages of Type = DATA, the ith bit is 1 (all others are 0) to indicate that this message carries the ith fragment. For messages of Type = SRR, the ith bit is set to 1 to indicate that the ith fragment has arrived, and it is set to 0 to indicate that the ith fragment is missing. Note that there are several ways to identify fragments. For example, the header could have contained a simple "fragment ID" field, with this field set to i to denote the ith fragment. The tricky part with this approach, as opposed to a bit-vector, is how the SRR specifies which fragments have arrived and which have not. If it takes an n-bit number to identify each missing fragment—as opposed to a single bit in a fixed-size bit-vector—then the SRR message will be of variable length, depending on how many fragments are missing. Variable-length headers are allowed, but they are a little trickier to process. On the other hand, one limitation of the BLAST header given above is that the length of the bit-vector limits each message to only 32 fragments. If the underlying network has an MTU of 1 KB, then this is sufficient to send up to 32-KB messages.