A Java Implementation of BLAST for a Networking Course

Jeffrey D. Martens
Department of Computer Science
Hood College
jmartens@cs.hood.edu

Abstract

The implementation of the bottom-layer of a simple RPC protocol stack is described and its possible use in introductory computer networking courses is described.

Introduction

In their introductory computer networking text, Peterson and Davie[2, pp. 405ff] introduce an RPC protocol stack: SELECT/CHAN/BLAST. The bottom layer, BLAST, provides end-to-end transport and message fragmentation. Although BLAST attempts to recover lost fragments, it does not ensure reliable transfer; this is left to CHAN. SELECT delivers messages to individual processes.

Given a facility such as BLAST, a student can implement various sorts of upper layers specified by an instructor or modify the BLAST implementation to perform in some different fashion.

Hood College's introductory networking course has a heavy programming emphasis, with the programming done using sockets in Java. Adding one or two assignments using BLAST would be compatible with this, and would illustrate a number of lower-level implementation details, both in terms of student-written code and in terms of instructor-written code read by the students, either as part of an assignment or as an example in lecture.

The BLAST Protocol

A general overview of BLAST will be given here. For details, readers are referred to Peterson and Davie[2].

BLAST, CHAN, and SELECT are intended to provide an example request-reply protocol for educational purposes. At the bottom layer, BLAST takes messages of up to 32KB and transfers them between hosts. Because many networks have a maximum transmission unit of much less than 32KB, BLAST fragments the messages into 1KB pieces. If at least one of these fragments of a message reaches the receiver, the receiver will request retransmission of missing fragments, assemble the fragments into a message, and pass the message to a higher layer.

There are three timers involved in the retransmit mechanism.

1. The `DONE` timers reside on the sender, with one associated with each outgoing message. The sender keeps a copy of each message for a short time in case the receiver requests retransmission of parts of the message. `DONE` is started when a message is sent, and restarted whenever a request regarding the message is received. When `DONE` expires, the message buffer is freed.

2. The `LAST_FRAG` timers reside on the receiver, with one associated with each received message. Once the fragments of a message begin to arrive, `LAST_FRAG`

is started. If it expires before all fragments arrive, a selective retransmission request (SRR) is returned to the sender, specifying which fragments to retransmit.

3. When `LAST_FRAG` expires, and also upon reception of the last fragment of a message, if there are missing fragments, the `RETRY` timer is started. Each time `RETRY` expires, an SRR is generated for any missing fragments. The receiver discards the message fragments if portions are still missing after three retries.

Timers are implemented as variables of type `long`, each holding the value of `System.currentTimeMillis()` at the instant the timer is started. The various timeout values are not specified by Peterson and Davie, and so are configurable in the implementation.

BLAST cannot handle messages above 32KB, but in a data transfer application the layer above BLAST is free to choose any message size up to 32KB. This has performance and memory overhead implications, which provide opportunities for student exercises. For example, attempting to transfer a sizable file using single-byte messages can overwhelm the *sender*, since the sender is required to maintain each message in memory until its DONE timer expires.

A BLAST Implementation

The author's BLAST implementation is provided as a Java package for inclusion in student programs. The author's implementation might be thought of as a reference implementation, since several concessions were made in favor of simplicity and readability at the cost of efficiency.

A reason to not use this as a reference implementation is that Peterson and Davie do not fully-specify BLAST. For example, bit numbering (from the left or from the right?) is not given, and they say that the last fragment of a message is marked, but do not say how. In this implementation, the last fragment is marked by setting the high bit in the `type` field, and this can be used to generate classroom discussion or as a test question.

BLAST is implemented as a multi-threaded singleton that sits on top of UDP. Students interface with BLAST via the `rpc.blast.Blast` class and its `send()` and `receive()` methods. A concession to the implementation is that, in addition to a protocol number and array of bytes, `send()` also requires the IP address and port number to identify the receiving process. `receive()` blithely assumes that any message received from any host is intended for the layer above, but it does report the sender's IP address to the upper layer.

As shown in figure one, the `Blast` class uses a `Sender` to send data and a `Receiver` to receive data. The `Sender` and `Receiver` each run as separate threads, with `Blast` itself running in the same thread as the calling layer. The three aforementioned classes are tied together with a `Pipe` instance, which provides FIFO communications between the threads. The `Sender` removes objects from the `Pipe`, which are inserted by `Blast` and by the `Receiver`. The inserted objects are messages to be transmitted. The `Receiver` sends SRRs.

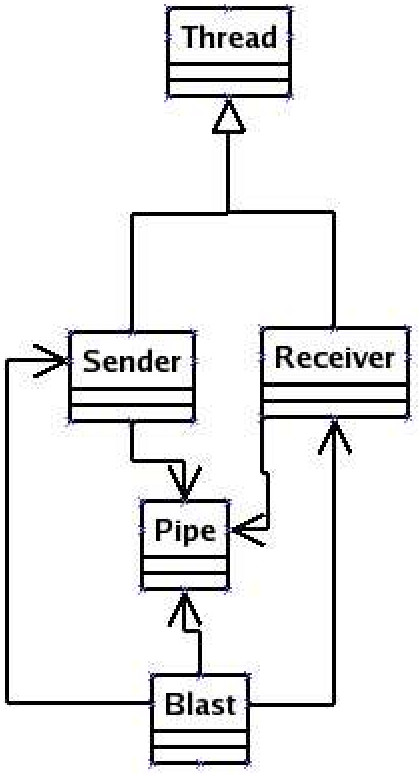

Figure One: Overview of Major Classes

The `rpc.blast.Blast` Class

The `Blast` class is a singleton, meaning that only one instance of the class exists within a process at a time; this restriction is enforced by the code. The UDP port at which it receives messages is configurable. The UDP port to which it sends messages is specified on a per-message basis by the upper layer, as is the destination IP address (actually, the `java.net.InetSocketAddress` class is used to specify both).

When `Blast` is started, it creates a `Pipe` (written by the author for an operating systems course, and described in [1]. `Blast` then creates and starts the `Sender` and `Receiver` threads, each of which is given a reference to the pipe. `Blast` and `Receiver` insert into the pipe. `Sender` reads from the pipe.

The layer above BLAST interacts with the `Blast` class predominantly via two methods, `send()` and `receive()`. To transmit, the upper layer calls `send()`. `send()` splits the request into fragments, and passes the fragments to the `Sender` class via the pipe. The transmission actually occurs within the `Sender` class.

`send()` makes local copies of parameters prior to returning to the upper layer. No references to the original parameter copies are kept within `Blast`. This avoids any chance of race conditions in cases where clients reuse objects.

The `receive()` method is passed a receive buffer and a protocol number from the upper layer, and returns a message matching the requested protocol number. If no such message is available, `receive()` blocks.

There are no restrictions on the size of the buffer. If the received message is larger than the buffer, the message is truncated. If the buffer is larger than the message, only the first part of it is used. `receive()` returns the number of bytes read, so the client knows how much of the buffer corresponds to the message. `receive()` also returns an `InetSocketAddress` in case the caller wishes to validate its correspondent.

If there are multiple clients in the layer above BLAST, each is honor-bound to identify itself using the correct protocol number. In other words, this implementation of BLAST trusts the layer above.

The general flow through `receive()` is that `Receiver.read()` is called to obtain a message with the desired protocol number. When the message is obtained, it consists of an array of fragments. These fragments are copies into the caller-supplied buffer.

The `rpc.blast.Sender` Class

The sender executes as a daemon thread. The Java virtual machine terminates once no non-daemon threads are executing. Thus, if the layer above terminates, the sender is not in itself sufficient to keep the virtual machine running.

Each message has to be kept until its DONE timer expires. Messages are held in `pending`, an instance of `java.util.Map`, which is accessed by message I.D. taken as a `java.lang.Integer`. Within `pending`, each I.D. maps to an instance of `PendingMessage`, which is a local private class, composed of a timestamp (DONE) and the message. `PendingMessage` also has a method, `hasExpired()`, which indicates to the sender whether the message can be deleted. Note that the local copy of the message cannot be deleted before DONE expires, since receivers are not expected to send acknowledgments, but rather retransmit requests. In principle, individual message fragments could be freed when an SRR is received, but this is not done.

In Java, thread execution begins in the `run()` method. In this implementation, `run()` is simple:

- block at the pipe waiting for something to do;
- send the object removed from the pipe to one of the the private `process()` methods. One of these processes `Message` instances, and the other processes `SelectiveRetransmit` instances.
  - `Message` instances are generated by `Blast.send()`.
  - `SelectiveRetransmit` instances are generated by `Receive.run()` in response to incoming SRRs.
- Call `checkTimers()`, which sweeps through all the pending messages in `pending`, looking for DONE timers that have expired.

The above three steps are wrapped in an infinite loop.

Checking for expired timers on every iteration of this loop is probably the simplest solution. Certainly, if no work is coming into the sender via the pipe, the sender's memory requirements are not increasing, and so it seems that this solution checks often enough. In conditions where many messages are being sent, however, one might argue that the timers are being checked too often, but these are also the conditions under which one would want to clean up as often as possible.

Checking the timers is straightforward. `checkTimers()` obtains an iterator with which it sweeps through the pending messages in the map, `pending`. Any expired message is removed.

Processing outgoing messages and processing SRRs differ for a few reasons. First, an outgoing message must have a unique message ID generated. The message ID associated with an SRR much match that of the partially-received message for which the local `Receiver` instance is missing fragments. Outgoing messages must be stored in `pending` for a time, whereas SRRs will not be re-sent, and so do not have to be kept.

The version of `process()` intended for outgoing messages sends each fragment in its own UDP datagram, encapsulates the message as a whole in a `PendingMessage`, and finally adds the pending message to `pending`.

The processing of outgoing SRRs is a bit more involved. First, the corresponding `PendingMessage` must be retrieved from `pending`. There is a possibility that the corresponding `PendingMessage` has been already been removed, in which case `process()` silently discards the request.

A mask within the SRR indicates which fragments have arrived successfully.

`process()` iterates through the bits of the mask, resending any fragments for which the corresponding bit is zero.

As mentioned above, the Peterson and Davie text is vague on how bits are numbered in the mask. This implementation numbers bits starting with zero as the least significant bit. It will inter-operate correctly only with other BLAST implementations that have made this choice.

If any fragments are re-sent, the DONE timer for the message must be restarted. If no fragments needed to be re-sent, e.g., if the BLAST on the other host were to use an SRR as an acknowledgment, then the pending message is freed.

Both versions of process actually call `Sender.send()` for transmission. The `send()` method itself is straightforward: given a fragment and a `SocketAddress`, it creates a `DatagramPacket` and sends via UDP.

`Sender` and its related classes are illustrated in figure two.

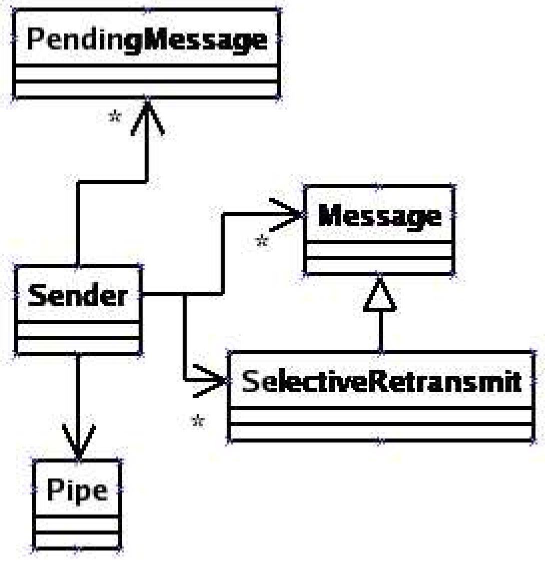

Figure Two: Classes important to the Sender

To review, the sender stores pending messages in a map, receives messages from `Blast` via the pipe, and receives SRR requests from the receiver via the pipe. The figure omits the `Fragment` class, which is used by `Sender`, `Message`, and `SelectiveRetransmit`.

The `rpc.blast.Receiver` Class

Like the sender, the receiver executes as a daemon thread. Its related classes are shown in figure three. PartialMessage contains the fragments of a message that has started to arrive, but is not yet complete. RecvTimerDaemon is a thread that periodically checks the LAST_FRAG and RETRY timers of all the partial messages, requesting retransmission and removing failed messages as each partial message's state warrants. This thread is started as a daemon by the Receiver constructor. As in the case of Sender, many of these classes make use of the Fragment class.

The major data structures with the Receiver class are ReadyMessages and reassembly. readyMessages is a list of messages for which all fragments have been received. It is ordered by age, oldest first, so when an upper layer requests a message, it will get the oldest with a particular protocol. The current implementation is fast if only one protocol is in use, but is a linear search, and so with multiple protocols, could be slow.

The reassembly structure is a map from message IDs to PartialMessage instances. This is used to store the fragments of messages as they arrive, and messages are moved from reassembly to readyMessages as they complete. Unfragmented (i.e., single-fragment) messages are never placed in reassembly.

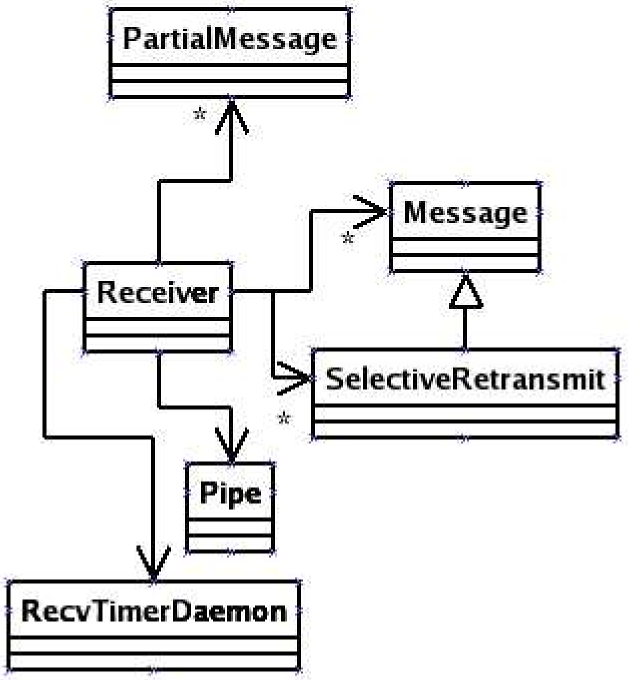

Figure Three: Classes important to the Receiver

The receiver's run() method is an infinite loop that reads UDP datagrams from the network, places each in a Fragment, and then handles it as either an SRR or an incoming data message. An incoming SRR is placed in a SelectiveRetransmit instance, and queued up in the pipe to the local Sender object.

An incoming data message is passed to `incomingData()`, which does a small amount of consistency checking, and silently discards any fragments that look bad. Then, if this is an unfragmented message, it is placed in `readyMessages`. Otherwise, we check to see if it is part of a message that's already in `reassembly`. If so, it's added to that message, and the message is checked to see if it's complete. If not, the new fragment is inserted into a new `PartialMessage`, which is added to `reassembly`. Finally, if this is the last fragment of a message, and some earlier fragments are missing, an SRR is generated.

At various points in the receiver and in the timer daemon, it is necessary to generate an SRR. This is done in a uniform fashion by invoking `generateSRR()`.

The other major responsibility of `Receiver` is to pass messages to `Blast.receive()`. This is done by `read()`, which scans `readyMessages` for a messages with the requested protocol number. If none is found, it waits until notified by `incomingData()` that another message has arrived. When it obtains an acceptable message, it removes the message from `readyMessages` and returns it.

This class has data that is shared across multiple threads: the `Receiver` thread itself, the receiver timer daemon, and the upper layer thread that indirectly enters through `read()`. As a result, access to `reassembly` and `readyMessages` is synchronized. Declaring entire methods to be synchronized carried a potentially significant performance penalty, and placing these containers in synchronized wrappers provides insufficient protection from race conditions. Thus, synchronized blocks are used. Access to `reassembly` is synchronized on `this`. Access to `readyMessages` and its contents are synchronized on `readyMessages`. To avoid deadlock, the "`this` lock" is always acquired prior to the "`readyMessages` lock."

Assignments using BLAST

There are a number of assignments that involve students writing code that makes direct use of the BLAST implementation described in this paper. Possibilities include:

- Students can measure performance of this BLAST implementation using different message sizes and different networks, e.g., dialup, broadband, and LANs.
- Students could produce a simple multiplexer that delivers messages to individual processes. This would be analogous to UDP or SELECT.
- Students could provide a reliable message-oriented service. This would be analogous to CHAN.
- Students could provide a reliable byte stream service. This would be analogous to TCP, though, presumably, greatly simplified.

Instructors should be aware that BLAST has no flow or congestion control, and so can stress a network.

As an alternative to writing a layer on top of BLAST, students could be asked to modify the behavior of BLAST in some manner, for example:

- BLAST uses selective retransmission requests. Students might be asked to instead acknowledge fragments individually (as in stop-and-wait) or cumulatively (as in TCP). Students could then be asked to compare the resulting performance with that of the original version.
- BLAST supports up to 32 fragments per message. The number of fragments supported could be increased by changing the 32-bit `FragMask` field to a fragment number field. This would change the way the receiver keeps track of received fragments, and the way it requests retransmissions from the sender.
- As mentioned above, the sender in this implementation of BLAST can be overwhelmed if asked to send many small messages quickly. The students could modify `Sender` so that it blocks if the size of the data structure, `pending`, is above a certain threshold.
- The author's implementation of BLAST is vulnerable to certain denial-of-service attacks. Students could be asked to close or tighten one or more of these holes.
- As mentioned above, `Blast` is a singleton. It might be interesting to allow multiple instances of `Blast`, but just one per incoming UDP port.
- It's possible for the sender to terminate before all messages are sent, or before the receiver on the other end has requested retransmissions of fragments. Students could augment the sender to provide status information so that an upper layer could wait for all pending messages to be resolved before exiting.
- The sender checks for expired timers once on each iteration of its main loop. Students might implement an alternative, such as:
  - The sender could, like the receiver, use a separate thread to check for timer expiration by sweeping through all the pending messages in regular intervals.
  - The sender could use a separate thread to check for timer expiration and order timers temporally. Then the thread could sleep until a timer is scheduled to expire, verify that it has indeed expired (the timer may have been restarted), free the resources, and then sleep till the next scheduled event.

  Students could also be asked to discuss these alternatives on an examination.
- The receiver's `readyMessages` structure could be modified to scale well with increasing numbers of protocols.

Many of the assignments suggested that modify this `rpc.blast` package involve skills and knowledge that are not specific to networking, and in some cases may seem to have little to do with networking. This in itself can be instructive, demonstrating to students the breadth of skills needed. But, many of these suggestions may not be appropriate for a networking course, and may not fit into other courses. One possibility of the sort that I have explored successfully in the past is that of allowing a good student to do an independent study in which `rpc.blast` is significantly modified and augmented.

The Future

This BLAST layer will be used for the first time in Hood's networking course during Spring semester, 2004. Documentation, source code, usage notes, and assignments will be available from the author's home page, at `http://cs.hood.edu/~jmartens/rpc/`.

References

[1] Martens, J. D. (2003). A Portable Thread API for Teaching Operating Systems. *The Journal of Computing Sciences in Colleges*, 18(6):119–125. Proceedings of the First CCSC Mid South Conference, Memphis, TN.

[2] Peterson, L. L. & Davie, B. S. (2003). *Computer Networks: a Systems Approach.* Morgan Kaufmann, San Francisco, third edition.