M.I.T. Laboratory for Computer Science

Request for Comments No. 217
March 18, 1982

BLAST, an Experimental File Transfer Protocol

by Dan Theriault

1. Introduction

BLAST is a file transfer protocol implemented using the User Datagram Protocol (UDP). It is an attempt to take advantage of bandwidth in high-speed local networks. The protocol was designed by David Reed. A prototype implementation has been written by Dan Theriault in Mesa 5.0 for the XEROX Alto. This document provides a sketchy description of the protocol, to disseminate the basic ideas and stimulate comments and further thought.

The underlying idea in the protocol is to minimize the number of round-trip delays in communication between hosts. Data packets to be transferred are transmitted unreliably to their destination. The receiver acknowledges full transmission of a file or requests retransmission of indicated packets.

A file is transferred in blocks of 512 8-bit data bytes. Only the last block of the file may contain fewer significant data bytes. These blocks are labelled with integers starting from zero. The convention used for indicating successfully transmitted blocks is a bit-vector in which a set n-th bit (where "n" is zero-based) implies a successfully transferred n-th data block of the file. The current maximum length of a file transferred by BLAST is 4096 blocks, which is slightly over 2 megabytes of data. A file of that size would make use of 512 bytes worth of bit-vector in the acknowledgement/retransmission-request packet.

The Blast protocol does not in any way interpret the contents of the data it transmits. This data is just sent as a collection of 8-bit bytes. It is assumed that the hosts will agree on the meaning of the transmitted data, and that a higher level protocol may filter or transform the data before sending or after receiving it using Blast. Blast is only intended to be a means of quickly getting bytes of data from one host to another.

The first word (2 bytes) of each Blast packet contains an opcode indicating the purpose of the packet. A value in the range [0..4096) indicates that the packet contains a data block, and indicates which block it is. A few of the remaining values indicate the various special packet types in the protocol. Packet formats are described in the appendix.

2. The Protocol

We will describe the protocol for a user sending to a server.

In brief, the scenario consists of:

1. A reliable exchange of packets in which the user indicates its wish to send a specified number of data bytes to the server for storage under a specified file name; and the server acknowledges, indicating a minimum delay the sender should allow between packets in order to allow time for storage.

2. Unreliable (unacknowledged) transmission by the sender of the entire sequence of data blocks, wrapped in self-identifying packets.

3. A reliable exchange of packets in which the sender indicates completion of data transmission; and the receiver either indicates successful reception of the entire sequence, or indicates which blocks it has and which blocks it has not successfully received.

4. If the receiver as successfully received all data, the transfer is complete. If not, the sender (unreliably) transmits the data blocks indicated as unreceived. GO TO 3.

As promised, a more detailed description follows.

2.1. UDP Connection

The User Datagram Protocol is connection-based. The user opens a connection with the server on port 71 (decimal), by convention.

2.2. Initial Handshake

The sender transmits a WishToSend packet to the receiver. This packet contains the number of bytes of data to be transmitted and the file name with which it is to be identified on the receiver's end. The sender then waits for a response from the receiver, retransmitting the packet if sufficient time elapses.

The receiver initially waits for packets. Upon receipt of a WishToSend, it begins preparations for the transfer, and responds with a StartSend packet. This packet indicates the minimum delay which the sender should allow between transmission of packets. This delay is merely an optimization to reduce the number of packets lost, if the receiver is slower than the sender (which is most often the case).

Upon receipt of a StartSend packet, the sender begins transmission of data.

2.3. Transmission of Data

First Pass

Each data block to be transmitted is, in turn, copied by the sender into a data packet whose opcode is the number associated with the block. Each packet is sent, with at least the inter-packet delay requested.

The receiver accepts all data packets and stores the data in an appropriate location. It maintains a bit-vector recording which blocks were successfully accepted.

Handshake After a Blast

After transmitting all data packets, the sender transmits a FinishedSending packet to indicate completion of the current pass, then waits for a response. It retransmits the FinishedSending packet when sufficient delay occurs.

Upon receipt of a FinishedSending packet, the receiver composes a Retransmit packet containing its bitvector indicating blocks successfully received and a new inter-packet delay. Alternatively, if it notices that it has received all blocks, it may send a ReceivedAll packet instead.

Subsequent Passes

Subsequent passes are quite similar to the first. The sender scans a bit-vector for zeroes indicating blocks needing transmission, and sends them off. Transmission terminates when the sender receives a ReceivedAll packet or a Retransmit packet with no significant zeroes.

Errors

The file transfer can be aborted by either the sender or the receiver by sending a FatalError packet, which contains a string indicating the problem.

3. Some Implementation Details

Diablo disk pages are 512 8-bit bytes long, which is the same size as the data blocks used in Blast.

Significant gains in speed can be gained with a few optimizations. As it reads pages from the file it is sending, the sender establishes a mapping from block number to disk address. On subsequent passes, it can make use of this information to form disk requests that read pages from the same cylinder during the same disk rotation. The receiver also does this to group its writes.

A. Packet Formats

WishToSend

sent by: Sender

contains:

```
        opcode          2 bytes
        nBytes          4 bytes
        fileName        >=1 bytes
```

where:

```
opcode = 177777 (octal).
nBytes = number of bytes in the file.
fileName = what the receiver should name the file.
        This is a string in the following format:
        1 byte          number, e.g. n, of characters in string
        n bytes         the ASCII characters in the string.
```

StartSend

sent by: Receiver

contains:

```
        opcode          2 bytes
        delay           2 bytes
```

where:

```
opcode = 177776 (octal)
delay = number of milli-seconds which sender should let elapse between
        transmission of packets.
```

Data

sent by: Sender

contains:

```
        opcode          2 bytes
        data            <=512 bytes
```

where:

```
opcode = J means this is the J-th block of the file (this is 0-based).
data = the data bytes composing the J-th block of the file.
        All blocks but the last will contain 512 bytes.
        The last may contain fewer.
```

FinishedSending

sent by: Sender

contains:

```
        opcode          2 bytes
```

where:

```
opcode = 177775 (octal)
```

Retransmit
sent by: Receiver
contains:

```
        opcode          2 bytes
        delay           2 bytes
        ack             <=512 bytes
```

where:

```
opcode = 177774 (octal)
delay = number of milli-seconds to allow between packets.
ack = a bit vector.
        N-th bit = 1 implies N-th block has been successfully transmitted.
```

ReceivedAll
sent by: Receiver
contains:

```
        opcode          2 bytes
```

where:

```
opcode = 177773 (octal)
```

FatalError
sent by: Sender, Receiver
contains:

```
        opcode          2 bytes
        reason          <=512 bytes
```

where:

```
opcode = 177772 (octal)
reason = a string (same format as the one in WishToSend)
```

WishToReceive
sent by: Receiver
contains:

```
        opcode          2 bytes
        fileName        <=512 bytes
```

where:

```
opcode = 177771 (octal)
fileName = a string (same format as the one in WishToSend).
        It contains the name of the file which the sender should transmit.
```