Network Working Group
Request for Comments: 1235

J. Ioannidis
G. Maguire, Jr.
Columbia University
Department of Computer Science

June 1991

The Coherent File Distribution Protocol

Status of this Memo

Introduction

The Coherent File Distribution Protocol (CFDP) has been designed to speed up one-to-many file transfer operations that exhibit traffic coherence on media with broadcast capability. Examples of such coherent file transfers are identical diskless workstations booting simultaneously, software upgrades being distributed to more than one machines at a site, a certain "object" (bitmap, graph, plain text, etc.) that is being discussed in a real-time electronic conference or class being sent to all participants, and so on.

In all these cases, we have a limited number of servers, usually only one, and <n> clients (where <n> can be large) that are being sent the same file. If these files are sent via multiple one-to-one transfers, the load on both the server and the network is greatly increased, as the same data are sent <n> times.

We propose a file distribution protocol that takes advantage of the broadcast nature of the communications medium (e.g., fiber, ethernet, packet radio) to drastically reduce the time needed for file transfer and the impact on the file server and the network. While this protocol was developed to allow the simultaneous booting of diskless workstations over our experimental packet-radio network, it can be used in any situation where coherent transfers take place.

CFDP was originally designed as a back-end protocol; a front-end interface (to convert file names and requests for them to file handles) is still needed, but a number of existing protocols can be adapted to use with CFDP. Two such reference applications have been developed; one is for diskless booting of workstations, a simplified

BOOTP [3] daemon (which we call sbootpd) and a simple, TFTP-like front end (which we call vtftp). In addition, our CFDP server has been extended to provide this front-end interface. We do not consider this front-end part of the CFDP protocol, however, we present it in this document to provide a complete example.

The two clients and the CFDP server are available as reference implementations for anonymous ftp from the site CS.COLUMBIA.EDU (128.59.16.20) in directory pub/cfdp/. Also, a companion document ("BOOTP extensions to support CFDP") lists the "vendor extensions" for BOOTP (a-la RFC-1084 [4]) that apply here.

Overview

CFDP is implemented as a protocol on top of UDP [5], but it can be implemented on top of any protocol that supports broadcast datagrams. Moreover, when IP multicast [6] implementations become more widespread, it would make more sense to use a multicast address to distribute CFDP packets, in order to reduce the overhead of non-participating machines.

A CFDP client that wants to receive a file first contacts a server to acquire a "ticket" for the file in question. This server could be a suitably modified BOOTP server, the equivalent of the tftpd daemon, etc. The server responds with a 32-bit ticket that will be used in the actual file transfers, the block size sent with each packet (which we shall call "BLKSZ" from now on), and the size (in bytes) of the file being transferred ("FILSZ"). BLKSZ should be a power of two. A good value for BLKSZ is 512. This way the total packet size (IPheader+UDPheader+CFDPheader+data=20+8+12+512=552), is kept well under the magic number 576, the minimum MTU for IP networks [7]. Note that this choice of BLKSZ supports transfers of files that are up to 32 Mbytes in size. At this point, the client should allocate enough buffer space (in memory, or on disk) so that received packets can be placed directly where they belong, in a way similar to the NetBLT protocol [8].

It is assumed that the CFDP server will also be informed about the ticket so that it can respond to requests. This can be done, for example, by having the CFDP server and the ticket server keep the table of ticket-to-filename mappings in shared memory, or having the CFDP server listening on a socket for this information. To reduce overhead, it is recommended that the CFDP server be the same process as the front-end (ticket) server.

After the client has received the ticket for the file, it starts listening for (broadcast) packets with the same ticket, that may exist due to an in-progress transfer of the same file. If it cannot detect any traffic, it sends to the CFDP server a request to start transmitting the whole file. The server then sends the entire file in small, equal-sized packets consisting of the ticket, the packet sequence number, the actual length of data in this packet (equal to BLKSZ, except for the last packet in the transfer), a 32-bit checksum, and the BLKSZ bytes of data. Upon receipt of each packet, the client checksums it, marks the corresponding block as received and places its contents in the appropriate place in the local file. If the client does not receive any packets within a timeout period, it sends to the CFDP server a request indicating which packets it has not yet received, and then goes back to the receiving mode. This process is repeated until the client has received all blocks of the file.

The CFDP server accepts requests for an entire file ("full" file requests, "FULREQ"s), or requests for a set of BLKSZ blocks ("partial" file requests, "PARREQ"s). In the first case, the server subsequently broadcasts the entire file, whereas in the second it only broadcasts the blocks requested. If a FULREQ or a PARREQ arrives while a transfer (of the same file) is in progress, the requests are ignored. When the server has sent all the requested packets, it returns to its idle state.

The CFDP server listens for requests on UDP/IP port "cfdpsrv". The clients accept packets on UDP/IP port "cfdpcln" (both to be defined by the site administrator), and this is the destination of the server’s broadcasts. Those two port numbers are sent to the client with the initial handshake packet, along with the ticket. If the minimal ticket server is implemented as described later in this document, it is recommended (for interoperability reasons) that it listens for requests on UDP/IP port 120 ("cfdptkt").

Let us now examine the protocol in more detail.

Protocol Specification

Initial Handshake (not strictly part of the protocol):

The client must acquire a ticket for the file it wishes to transfer, and the CFDP server should be informed of the ticket/filename mapping. Again, this can be done inside a BOOTP server, a modified TFTP server, etc., or it can be part of the CFDP server itself. We present here a suggested protocol for this phase.

The client sends a "Request Ticket" (REQTKT) request to the CFDP Ticket server, using UDP port "cfdptkt". If the address of the server is unknown, the packet can be sent to the local broadcast address. Figure 1 shows the format of this packet.

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |      'R'      |      'Q'      |      'T'      |      'K'      |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                                                               |
    /                                                               /
    \     Filename, null-terminated, up to 512 octets               \
    /                                                               /
    |                                                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 1: "ReQuest TicKet" packet.

The filename is limited to 512 octets. This should not cause a problem in most, if not all, cases.

The ticket server replies with a "This is Your Ticket" (TIYT) packet containing the ticket. Figure 2 shows the format of this packet.

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |      'T'      |      'I'      |      'Y'      |      'T'      |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                            "ticket"                           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                       BLKSZ (by default 512)                  |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                             FILSZ                             |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |            IP address of CFDP server (network order)          |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |   client UDP port# (cfdpcln)  |   server UDP port# (cfdpsrv)  |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 2: "This Is Your Ticket" packet.

The reply is sent to the UDP port that the RQTK request came from. The IP address of the CFDP server is provided because the original handshake server is not necessarily on the same machine as the ticket server, let alone the same process. Similarly, the cfdpcln and cfdpsrv port numbers (in network order) are communicated to the client. If the client does not use this ticket server, but rather uses BOOTP or something else, that other server should be responsible for providing the values of cfdpcln and cfdpsrv. The ticket server also communicates this ticket/filename/filesize to the real CFDP server. It is recommended that the ticket requests be handled by the regular CFDP server, in which case informing the CFDP server of the ticket/filename binding is trivial (as it is internal to the process).

Once the client has received the ticket for the filename it has requested, the file distribution can proceed.

Client Protocol:

Once the ticket has been established, the client starts listening for broadcast packets on the cfdpcln/udp port that have the same "ticket" as the one it is interested in. In the state diagram below, the client is in the CLSTART state. If the client can detect no packets with that ticket within a specified timeout period, "TOUT-1", it assumes that no transfer is in progress. It then sends a FULREQ packet (see discussion above) to the CFDP server, asking it to start transmitting the file, and goes back to the CLSTART state (so that it can time out again if the FULREQ packet is lost). Figure 3 shows the format of the FULREQ packet.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            "ticket"                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            checksum                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'F'      |       0       |         length == 0           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 3: FULREQ (FULl file REQuest) packet.

When the first packet arrives, the client moves to the RXING state and starts processing packets. Figure 4 shows the format of a data packet.

```
        0                   1                   2                   3
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                            "ticket"                           |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                            checksum                           |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |          block number         |          data length          |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                                                               |
       /                                                               /
       \      up to BLKSZ octets of data                               \
       /                                                               /
       |                                                               |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 4: Data Packet

The format is self-explanatory. "Block number" the offset (in multiples of BLKSZ) from the beginning of the file, data length is always BLKSZ except for the very last packet, where it can be less than that, and the rest is data.

As each packet arrives, the client verifies the checksum and places the data in the appropriate position in the file. While the file is incomplete and packets keep arriving, the client stays in the RXING state, processing them. If the client does not receive any packets within a specified period of time, "TOUT-2", it times out and moves to the INCMPLT state. There, it determines which packets have not yet been received and transmits a PARREQ request to the server. This request consists of as many block numbers as will fit in the data area of a data packet. If one such request is not enough to request all missing packets, more will be requested when the server has finished sending this batch and the client times out. Also, if the client has sent a PARREQ and has not received any data packets within a timeout period, "TOUT-3", it retransmits the same PARREQ. Figure 5 shows the format of the PARtial REQuest packet.

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                            "ticket"                           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                            checksum                           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |      'P'      |       0       |      data length (2*N)        |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |           Block #0            |           Block #1            |
    |           Block #2            |           Block #3            |
    /                                                               /
    \      data  (block numbers requested)                          \
    /                                                               /
    |           Block #N-2          |           Block #N-1          |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 5: PARREQ (PARtial file REQuest) packet.

When all packets have been received the client enters the CLEND state and stops listening.

Figure 6 summarizes the client's operations in a state diagram.

```
                           +-----------+
                           |  CLSTART  |
                           |           | <---.
                           |   send    |     | timeout TOUT-1
                           |  FULREQ   | ----'
                           |           |
                           +-----------+
                                 |
             received packet     | received packet
      .-----------------------.  |
      |                       V  V
     +---------+             +---------+
     | INCMPLT |             |  RXING  |
     |         |   timeout   |         | <---.
     |  send   |<------------| process |     | received packet
     | PARREQ  |    TOUT-2   | packet  | ----'
     |         |             |         |
     +---------+             +---------+
        ^   |                    |
        |   |                    |finished
        '---'                    |
       timeout                   V
        TOUT-3               +---------+
                             |  CLEND  |
                             +---------+
```

Fig. 6: Client State Transition Diagram

Server Protocol:

As described above, the CFDP server accepts two kinds of requests: a request for a full file transfer, "FULREQ", and a request for a partial (some blocks only) file transfer, "PARREQ". For the first, it is instructed to start sending out the contents of a file. For the second, it will only send out the requested blocks. The server should know at all times which files correspond to which "tickets", and handle them appropriately. Note that this may run into implementation limits on some Unix systems (e.g., on older systems, a process could only have 20 files open at any one time), but that should not normally pose a problem.

The server is initially in the SIDLE state, idling (see diagram below). When it receives a FULREQ packet, it goes to the FULSND state, whence it broadcasts the entire contents of the file whose ticket was specified in the FULREQ packet. When it is done, it goes back to the SIDLE state. When it receives a PARREQ packet, it goes to the PARSND state and broadcasts the blocks specified in the PARREQ packet. When it has finished processing the block request, it goes once again back to the SIDLE state.

```
                     receive    +-------+    receive
                .---------------| SIDLE |---------------.
                |    FULREQ     +-------+     PARREQ    |
                |                 ^   ^                 |
                |                 |   |                 |
                V                 |   |                 V
            +--------+            |   |            +--------+
            | FULSND |            |   |            | PARSND |
            |        |    done    |   |    done    |        |
            |  send  |------------'   '------------|  send  |
            | entire |                             | req'ed |
            |  file  |                             | blocks |
            +--------+                             +--------+
```

Fig. 7: Server State Transition Diagram

Packet Formats

The structure of the packets has been already described. In all packet formats, numbers are assumed to be in network order ("big-endian"), including the ticket and the checksum.

The checksum is the two's complement of the unsigned 32-bit sum with no end-around-carry (to facilitate implementation) of the rest of the packet. Thus, to compute the checksum, the sender sets that field to zero and adds the contents of the packet including the header. The it takes the two's complement of that sum and uses it as the checksum. Similarly, the receiver just adds the entire contents of the packet, ignoring overflows, and the result should be zero.

Tuneable Parameters: Packet Size, Delays and Timeouts

It is recommended that the packet size be less than the minimum MTU on the connected network where the file transfers are taking place. We want this so that there be no fragmentation; one UDP packet should correspond to one hardware packet. It is further recommended that the packet size be a power of two, so that offsets into the file can be computed from the block number by a simple logical shift operation. Also, it is usually the case that page-aligned transfers are faster on machines with a paged address space. Small packet sizes are inefficient, since the header will be a larger fraction of the packet, and packets larger than the MTU will be fragmented. A good selection for BLKSZ is 512 or 1024. Using that BLKSZ, one can transfer files up to 32MB or 64MB respectively (since the limit is the 16-bit packet sequence number). This is adequate for all but copying complete disks, and it allows twice as many packets to be requested in a PARREQ request than if the sequence number were 32 bits. If larger files must be transferred, they could be treated as multiple logical files, each with a size of 32MB (or 64MB).

Since most UDP/IP implementations do not buffer enough UDP datagrams, the server should not transmit packets faster than its clients can consume them. Since this is a one-to-many transfer, it is not desirable to use flow-control to ensure that the server does not overrun the clients. Rather, we insert a small delay between packets transmitted. A good estimate of the proper delay between two successive packets is twice the amount of time it takes for the interface to transmit a packet. On Unix implementations, the ping program can be used to provide an estimate of this, by specifying the same packet length on the command line as the expected CFDP packet length (usually 524 bytes).

The timeouts for the client are harder to compute. While there is a provision for the three timeouts (TOUT-1, TOUT-2 and TOUT-3) to be different, there is no compelling reason not to make them the same. Experimentally, we have determined that a timeout of 6-8 times the transfer time for a packet works best. A timeout of less than that runs the risk of mistaking a transient network problem for a timeout, and more than that delays the transfer too much.

Summary

To summarize, here is the timeline of a sample file distribution using CFDP to three clients. Here we request a file with eight blocks. States are capitalized, requests are preceded with a ’<’ sign, replies are followed by a ’>’ sign, block numbers are preceded with a ’#’ sign, and actions are in parentheses:

SERVER	CLIENT1	CLIENT-2	CLIENT-3	comments
IDLE				everybody idle
	CLSTART			CL1 wants a file
	<TKRQ			requests ticket
TIYT>				server replies
	(timeout)			listens for traffic
	<FULREQ			full request
#0	RXING			CL1 starts receiving
	(rx 0)			
#1	(rx 1)	CLSTART		CL2 decides to join
		<TKRQ		
#2	(rx 2)			SRV still sending
TIYT>				responds to TKRQ
#3	(rx 3)	(listens)		CL2 listens
		RXING		found traffic

```
#4            (rx 4)      (rx 4)        CLSTART      CL3 joins in
                                        <TKRQ
#5            (missed)    (rx 5)                     CL1 missed a packet
TIYT>                                   (listens)
#6            (rx 6)      (rx 6)        RXING        CL3 found traffic

#7            (rx 7)      (rx 7)        (rx 7)       Server finished
IDLE
              (wait)      (wait)        (wait)       CL1 managed to
              (timeout)   (wait)        (wait)       timeout
              <PARREQ[5]  (timeout)     (timeout)    CL1 blockrequests...
#5            (rx 5)      <PARREQ[0123] <PARREQ[0123456] ignored by SRV
              CLEND                                  CL1 has all packets
IDLE                      (wait)        (wait)       CL2+3 missed #5
                          (timeout)     (timeout)
                          <PARREQ[0123] <PARREQ[0123456] CL2's req gets
#0                        (rx 0)        (rx 0)       through, CL3 ignored
#1                        (rx 1)        (rx 1)       moving along
#2                        (rx 2)        (rx 2)
#3                        (rx 3)        (rx 3)
IDLE                      CLEND         (wait)       CL2 finished
                                        (timeout)
                                        <PARREQ[456]
#4                                      (rx 4)
#5                                      (rx 5)
#5                                      (rx 6)
IDLE                                    CLEND        CL3 finished
```

References

[1] Sollins, K., "The TFTP Protocol (Revision 2)", RFC 783, MIT, June 1981.

[2] Finlayson, R., "Bootstrap Loading Using TFTP", RFC 906, Stanford, June 1984.

[3] Croft, W., and J. Gilmore, "Bootstrap Protocol", RFC 951, Stanford and SUN Microsystems, September 1985.

[4] Reynolds, J., "BOOTP Vendor Information Extensions", RFC 1084, USC/Information Sciences Institute, December 1988.

[5] Postel, J., "User Datagram Protocol", RFC 768, USC/Information Sciences Institute, August 1980.

[6] Deering, S., "Host Extensions for IP Multicasting", RFC 1112, Stanford University, August 1989.

[7] Postel, J., "Internet Protocol - DARPA Internet Program Protocol Specification", RFC 791, DARPA, September 1981.

[8] Clark, D., Lambert, M., and L. Zhang, "NETBLT: A Bulk Data Transfer Protocol", RFC 998, MIT, March 1987.

Security Considerations

Security issues are not discussed in this memo.

Authors' Addresses

John Ioannidis
Columbia University
Department of Computer Science
450 Computer Science
New York, NY 10027

EMail: ji@cs.columbia.edu

Gerald Q. Maguire, Jr.
Columbia University
Department of Computer Science
450 Computer Science
New York, NY 10027

Phone: (212) 854-2736

EMail: maguire@cs.columbia.edu