XEROX **Palo Alto Research Center**

June 15, 1976

To: Distributed Computing

From: John Shoch

Subject: **EFTP: A PUP-based Ether File Transfer Protocol**

file: <pup>eftpspec.press

This document describes a means for simply moving files between two computers attached to a packet switched network. Known historically as the Experimental Ether File Transfer Protocol (and now called simply EFTP), this mechanism can be used in a stand-alone program, or as a program module embedded within other software systems. While the protocol is not adequate for attacking the problems of more general file transfers, it does lead to an economical and reliable implementation.

1. Companion Documents

Background information on the Ethernet, PUPs, and the Byte Stream Protocol (BSP):
---Bob Metcalfe, *PUP Specification*, dated October 20, 1975.
---Bob Metcalfe and David Boggs, *Ethernet: Distributed Packet Switching for Local Computer Networks*, November 1975 [to appear in CACM, 1976].
---Ed Taft, *Implementation of Pup in Tenex*, dated October 28, 1975.
Detailed description of a more general File Transfer Protocol:
---John Shoch, *A File Transfer Protocol Using the BSP -- 2nd edition*, May 19, 1976.

2. Assumptions and Basic Principles

---A general mechanism for transferring files is described in the specification for the full File Transfer Protocol. The principle characteristic of that design is the notion of a machine-independent *file property list*, supporting transfers among machines with very different characteristics and operating system conventions. The current EFTP design (derived from the original Experimental Ethernet File Transfer Protocol) does not provide all of those facilities, but the simplified protocol makes it easier to construct a separate file transfer module, which can then be used within other software systems.

---An outline of the protocol can be found in the paper by Metcalfe and Boggs, cited above; although some important changes and improvements have been made.

---Instead of using only Ethernet packets, the protocol now uses the more general definition of a uniform packet, the PUP. The major impact of this change is to extend the effective addressing capability of the protocol in order to communicate with hosts connected to other networks (and which must be reached through an inter-network gateway).

---File transfers will generally take place between hosts; there is no attempt to support multiple file transfers at one time from one machine. For simplicity, the *receiver* will also be the *server*, and the *sender* of data will also be the *user*. Thus, unattended servers (such as a printing host) can receive files; but it is not meaningful for an unattended user process to attempt to send a file. There is, therefore only a degenerate form of rendezvous: a user may launch packet 0 to socket 20 of the foreign host; the server may choose to acknowledge it, implicitly opening the connection.

---The contents of a file move as an unnamed collection of bytes; the receiver must be prepared to name the file, or dispose of it appropriately.

3. Details of the Protocol

---There are only four packet types used:

EFTPData
EFTPAck
EFTPAbort
EFTPEnd

These are all registered PUP types, with the value carried in the PUP-type field. The length field of the PUP is the byte count of the data, plus the PUP overhead. Note that the length may be odd, in which case the PUP carries one byte of garbage at the end of the data section, which should be ignored.

---Each data packet contains a sequence number, starting at 0. The sequence number is carried in the low order 16 bits of the id field of the PUP (id^2, in PUP parlance).

---There may not be more than one outstanding sequence number at any time; the sender must receive a proper acknowledgement for packet *n* before he attempts to send packet *n+1*. If a proper ack is not returned, the sender should eventually timeout and retransmit. If an ack should get lost, a receiver may get another copy of the previous packet; the data it contains should not be re-processed, but the packet itself should be acknowledged appropriately.

---After transmitting the last data packet, and receiving an appropriate acknowledgment, the sender will transmit an EFTPEnd, with the next sequence number.

---The receiver should verify this packet, and acknowledge it accordingly, but that is not sufficient. It is possible that the receiver would send the ack and then depart. Should this ack be lost, the sender would re-transmit the EFTPEnd. Since the receiver is no longer listening, the sender would never get a response, and would not be able to verify that the connection had been ended properly.

---To handle this situation, a ’three-way handshake’ takes place to terminate the connection.

1. The sender transmits an EFTPEnd, in sequence.

2. The receiver responds with an EFTPAck, but instead of departing he remains listening, performing a *dally*. The receiver knows that the file has been received completely, but this extra dally period is done as a favor to the sender: should he fail to get the Ack and then re-transmit, the receiver will still be there to generate another Ack. Thus, the dally period should be long compared to the expected retransmission rate of the sender.

3. When the sender finally receives the Ack for the first EFTPEnd, he now also knows that the transaction has been completed. It is possible, however, that the receiver is still dallying at the other end; as a reciprocal favor, the sender now transmits another EFTPEnd, with the next sequence number, to release the receiver from the dally period. This packet is not acknowledged, and will not be re-transmitted. In the best case, the receiver will get the second End, and promptly depart; in the worst case, the receiver will remain until the end of the dally period, and then depart, still secure in the knowledge that the whole file was received. [It is worth noting that there is a finite probability that this scheme can fail: if the

sender transmits the first EFTPEnd, misses the Ack, but then cannot retransmit until after the receiver has finished dallying, he will never get the Ack. (This might happen if the sender were interrupted, and was not able to respond for a lengthy period of time.) This probability can be kept arbitrarily small, however, by insuring that the receiver's dally period is lengthy compared to the expected retransmission interval of the sender.]

4. Aborts, Resets, and other messages

It is hoped that this straightforward protocol can be modified slightly to provide one valuable service: allow an automatic server to receive more than one file at a time (this would be very useful in a network printing server, for example). To support this, several abort and reset mechanisms have been provided.

Packets of type EFTPAbort carry a machine readable abort code, as well as a human readable abort string: the abort code is carried in the first word of the data field, and the optional string therefore begins at byte 3. The length of the string is therefore equal to:

puplength - (pupOvBytes + 2)

A. Externally generated aborts

Certain external conditions may cause a participant in a file transfer to request an abort: the user/sender makes a mistake, the server/receiver realizes that the disk has become full, etc. If either party is intending to drop a connection before the transfer is complete, it is considered good manners to launch an abort before leaving. There are two general codes provided (users are encouraged to provide more specific textual error messages, whenever possible):

ExternalSenderAbort
ExternalReceiverAbort

B. Internally generated aborts

The EFTP mechanism may detect certain unusual conditions which require an abort or a reset.

1. While receiving one file, the server/receiver might refuse to respond at all to a packet from another source, and turn a deaf ear. Instead, the server will generate a non-fatal abort message, asking the second server to wait and try again soon. Thus, the second user might chose to wait longer before giving up. [Note that the server program must be able to recognize the second incoming request, record the fact, and generate an appropriate Abort.] This would be a ReceiverBusyAbort.

2. Should the server crash and restart, it may find itself receiving packets from a user who was in the midst of a transaction. In this case the server will generate an OutOfSynchAbort, telling the user that synchronization has been lost; the user may attempt to restart, or merely report a failure.

3. Should the user crash and start sending again, the server might suddenly find that packet 0 is mysteriously arriving again. Rather than sending an Abort, this retransmission of packet 0 (from the current foreign host) is taken as a request to restart the file transfer. The server/receiver can then reset the file, and begin again. Given this feature, a user may explicitly decide that something is wrong and request a reset by starting to transmmit from 0 again. [This scheme would break down, of course, if there were ever 2^16 packets in one transfer; but that is more data than can fit on most of our storage devices....] Note that this reset does not require transmission of an EFTPAbort packet.

5. A final note

This protocol may be subject to substantial change, as the Reliable Packet Protocol becomes more concrete.

Appendix: Assignment of packet types and abort codes

1. EFTPReceiveSocket

By convention, the EFTP receiver/server will run on socket #20.

2. PUP types:

EFTPData	= #30	
EFTPAck	= #31	
EFTPEnd	= #32	
EFTPAbort	= #33	[known, of course, as *error 33*....]

3. Abort codes (to be embedded within an EFTPAbort packet)

ExternalSenderAbort	= 1
ExternalReceiverAbort	= 2
ReceiverBusyAbort	= 3
OutOfSynchAbort	= 4