AT&T Technical Journal
Vol. 64, No. 10, December 1985
Printed in U.S.A.

A New File Transfer Protocol

By S. AGGARWAL,* K. SABNANI,* and B. GOPINATH†

(Manuscript received November 15, 1984)

We describe a new File Transfer Protocol (FTP) that provides a simple and efficient way of transferring files between heterogeneous systems, such as the *UNIX*™ operating system, Duplex Multi-Environment Real-Time (DMERT), and IBM/MVS. This FTP has been adopted as an AT&T standard. In this protocol, global functions requiring close coordination are separated from local functions. Functions that require global coordination are mandatory parts of the protocol and must be implemented uniformly. Local functions, such as file management and user interface, can be adjusted to local needs and might even be optional. This results in a flexible protocol that can be implemented at various levels of complexity. Thus, FTP implementations can range from very simple ones that provide basic file transfer service to highly complex ones that provide extensive security checking and allow a variety of file management services.

I. INTRODUCTION

File transfers typically represent a large percentage of the traffic volume on data networks. In one internal AT&T network, for example, over 50 percent of the data volume is file transfer traffic. Thus, it is important that file transfers be implemented through an efficient, flexible, and reliable protocol.

In the past, several customized protocols have been developed by various groups within AT&T. Each protocol was designed with only one application in mind, limiting its applicability and functionality. In this paper, we describe a new file transfer protocol that provides

* AT&T Bell Laboratories. † Bell Communications Research, Inc.

the functionality of all file transfer protocols previously used within AT&T. This protocol, called the BX.25 File Transfer Protocol (or simply the FTP), provides a simple and efficient way to transfer files between heterogeneous systems such as the *UNIX* operating system, Duplex Multi-Environment Real-Time (DMERT),[1] IBM/MVS, and UNIVAC/EXEC. This FTP has been adopted as an AT&T standard.[2] Several groups in the company have implemented or are implementing this protocol.[3]

Our approach in designing the FTP was to separate global functions requiring close coordination (such as parameter negotiation) from functions that could be done locally at each individual node. Functions that require global coordination are mandatory parts of the protocol and must be implemented uniformly. Local functions, such as file management and user interface, can be adjusted to local needs and might even be optional. This results in a flexible protocol that can be implemented at various levels of complexity. One can view the FTP as essentially an intelligent bulk data transfer facility.

The FTP is a three-party protocol in which a user at a remote node (the initiator) can transfer files between a source and a destination node. The FTP offers three types of services: Copy, Cancel, and Status. The Copy service allows the transfer of a set of files between the source and destination nodes. The Cancel and Status services, respectively, allow the user to cancel a transfer and request information about a transfer. All services can be done in the background, requiring minimal user supervision.

Our approach in specifying the FTP is consistent with the one currently recommended by the protocol community. This approach recommends that one first describe the services offered by the protocol to the upper layer, as well as the services required by the protocol from the lower layer. Then, it defines the peer-level protocol that fills the gap between the upper and lower layers. Typical protocol specifications, such as levels 2 and 3 of X.25,[4] cover only peer-level message exchange rules and formats.

The application program resident at each node that performs file transfers will be called a File Transfer System (FTS). Functions performed by the FTS cover the application and presentation layer functions as defined in the Open System Interconnection (OSI) model.[5]

As shown in Fig. 1, an FTS has several interfaces: an interface with the user, a data transfer facility interface, and interfaces with the file system and with the operating system. In addition, an FTS communicates with the remote FTSs using a peer-level protocol. The user is a person or a computer program that wants to use the file transfer service. The data transfer facility is the lower level of protocol, such

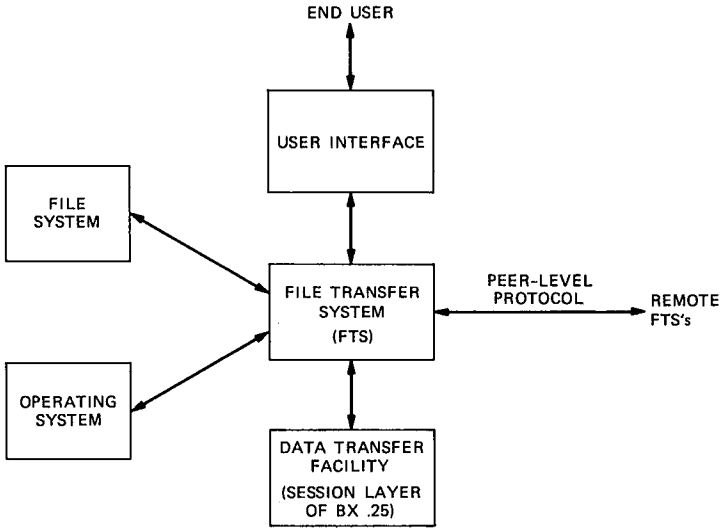


Fig. 1—Interfaces for the file transfer system in any node.

as the BX.25 session layer. The file system is the local storage system for files, and the operating system manages the local computing resources.

The user issues a command to the local FTS to perform a file transfer. The local FTS checks the command for its syntactic correctness. It reads or writes the files being transferred using the local file system. It sends or receives file data to/from a remote FTS through the data transfer facility. The FTS invokes local processing (see pre- and postprocessors below) on the file contents using the operating system interface.

In Section II, we describe our overall design approach in greater detail. We then describe services offered by the FTP in Section III, and identify the services required from the data transport layer in Section IV. The peer-level protocol is described in Section V. Section VI describes the negotiation procedure during which the source FTS and the destination FTS agree on the options for file transfers. Some examples are given in Section VII. We also describe a formal specification of the FTP using the selection/resolution model in Section VIII.[6,7] This specification gives a more complete and precise description of the protocol than an English language specification alone can provide. In Section IX, we compare the FTP with others reported in the literature. Finally, we make some concluding comments in Section X.

II. DESIGN APPROACH

In our design, we have clearly separated functions requiring global

coordination from those that are local to a node. Every implementation would be required to implement the global coordination functions that would form the core of the protocol. Implementors would be free to implement the local functions as needed. Implementations would thus range from very simple ones that provide basic file transfer services to highly complex ones that provide extensive security checking, allow a variety of file management capabilities, and provide a sophisticated user interface. Furthermore, the flexibility in implementing local functions results in a file transfer protocol that works in a heterogeneous environment, is adaptable to diverse local needs, and can evolve as requirements change.

The core of the protocol implements the coordination functions to transfer a byte stream efficiently. Translation between the file formats at the source and the destination is done outside the core of the protocol by local functions called preprocessors and postprocessors.

A preprocessor is a local function at the source that translates the structure and the contents of the file being transferred to a byte stream. Similarly, a postprocessor at the destination translates the byte stream back to the file at the destination. The use of these local functions provides a great deal of flexibility and allows implementations to expand gracefully.

For example, if the initial requirements are file transfers between a *UNIX* system and an IBM/MVS system, we need only write pre- and postprocessors for translation between their file formats. As additional systems, such as a Honeywell computer, are added, we will then add new pre- and postprocessors.

The FTP does not have a networkwide standard for file naming. A user must supply all the naming information required for accessing a remote file. This includes the information about the device name on which the file is stored, as well as the path name of the file. This approach allows us to keep the FTP core small in size.

The FTP uses checkpointing to transfer a byte stream efficiently. During a file transfer, checkpoints are set along the way. After recovery from a transmission failure, the file transfer is resumed from a checkpoint up to which the destination has received the file data correctly. The file transfer does not have to be restarted from the beginning. This feature is useful when transferring very large files such as those stored on several magnetic tapes.

Our security policy makes it difficult for a user to break into the security mechanism of a remote file system. To access a remote file, the user must provide access information required at the remote file system. The user without the proper access information is not allowed to access a remote file. In the following sections, we give further details of the protocol.

III. FTP SERVICE SPECIFICATION

In this section, we describe the services offered by the FTP: Copy, Cancel, and Status. This set of services is sufficient for most file transfer applications. For each service, all the information described here must be provided to the local FTS. We describe here the interaction between a user and an FTS while providing each one of these services.

To invoke a service, the user must issue a command to the local FTS. The syntax of the command is a local decision. However, the information contained in the command is required for global coordination. Thus, in any implementation of the FTP, each command must contain the information given below.

3.1 Copy service

For the Copy service, the command must contain the following information:

COPY ⟨User id⟩ ⟨Source address⟩
⟨File descriptor at the source⟩ ⟨Preprocessors⟩
⟨Destination address⟩
⟨File descriptor at the destination⟩ ⟨Postprocessors⟩
⟨Priority⟩

where

⟨User id⟩ identifies the user initiating the file transfer,

⟨Source address⟩ identifies the computer on which the files to be transferred reside,

⟨File descriptor at the source⟩ is the list of file names to be transferred,

⟨Pre-processors⟩ is a list of programs executed on the files before transfer,

⟨Destination address⟩ identifies the computer to which the files are to be transferred,

⟨File descriptor at the destination⟩ is the list of file names at the destination. The number of files in this list must be the same as that in the list ⟨File descriptor at the source⟩,

⟨Post-processors⟩ is a list of programs executed at the destination on the files transferred,

⟨Priority⟩ specifies the priority level assigned to the copy command. This field specifies how the FTS should treat the present job compared to the other file transfers waiting to be done at the local FTS.

The initiator FTS checks the syntactic correctness of the copy command. The number of file names in the list ⟨File descriptor at the source⟩ must be the same as that in ⟨File descriptor at the destination⟩. If there is an error in the copy command, then the FTS returns

an error message to the user. Otherwise, the FTS returns a job number to the user. The job number identifies a copy command throughout the network and the command's lifetime. The format for a job number ensures that it is unique throughout the network. The job number has the following format:

⟨Initiator address⟩ ⟨Unique number assigned by the initiator⟩

where

⟨Initiator address⟩ specifies the address of the initiator,

⟨Unique number assigned by the initiator FTS⟩ is a number assigned to the command; it uniquely identifies the command at the initiator.

The local FTS must notify the user upon successful or unsuccessful completion of the file transfer.

3.2 Status service

A user at the initiator FTS can inquire about the status of a file transfer request using the following command:

STATUS ⟨Job number⟩.

The status of a file transfer request must provide the user with the following information:

- Whether preprocessing, file transfer, and postprocessing have started, and, if they have, whether they have been completed successfully or unsuccessfully;
- If unsuccessful, the cause of the error.

Additional information, such as percentage of transfer completed, could also be provided to the user. At the initiator, the FTS returns the status described above to the user.

3.3 Cancel service

In a similar manner, a user at the initiator can cancel a file transfer in progress. For such a service, a user issues the following command:

CANCEL ⟨Job number⟩.

The initiator FTS must authenticate the user's identity and check the user's privileges. If the user is not allowed to cancel, then the initiator FTS must return an error message to the user. Otherwise, the file transfer must be canceled. In addition, the state of files modified as a result of the file transfer request must be restored to what it was before the carrying out of the file transfer request. The services described are provided using the peer-level protocol in Section V.

IV. DATA TRANSPORT SERVICES REQUIRED

The FTP requires certain data transport services from the lower

layer. More specifically, it requires a connection-oriented service. In addition, a connectionless service is desirable for short message exchanges. Both data transport services are provided by the BX.25 Session Layer.[4] In this section, we describe these two data-transport services.

4.1 Connection-oriented service

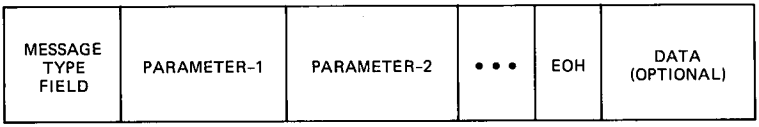
The connection-oriented service must transfer messages of variable length from a transmitter FTS* to a receiver FTS in error-free form and in the same sequence as originally delivered to the transmitter FTS. If it is unable to deliver messages in sequence because of a network failure, it must inform the transmitter FTS. Once a connection is set up between two FTSs, it can be used to transport the files from several file transfer requests. The following service primitives, or their equivalent, must be provided to the FTS:

1. *Connect* (*Destination Address, FTS*). Using the *Connect* primitive, the FTS in the local computer establishes a connection with an FTS resident in the computer with the address *Destination Address.* If the connection is successfully established, the data transport entity returns to the FTS *Connection id,* a unique identifier. The FTS uses *Connection id* later to reference the connection just established. If the data transport entity is unable to establish the connection, it notifies the local FTS about the failure.
2. *Send* (*Connection id, message*). This primitive is used to send a block of data called *message* on the connection *Connection id.*
3. *Receive* (*FTS, message, Connection id*). The local FTS uses the service primitive *Receive* to receive a block of data *message* from the connection *Connection id.*
4. *Disconnect* (*Connection id*). The FTS uses the *Disconnect* service primitive to break the connection *Connection id.* The data transport facility delivers any messages in transit before the breakup.
5. *Abort* (*Connection id*). This primitive is used to abnormally break up the connection *Connection id.* The data transport facility does not ensure delivery of the messages in transit before the breakup.

4.2 Connectionless service

A Connectionless service is useful for the transport of single messages between FTSs. It ensures that such message exchanges are done with minimal network resources. Establishment and termination of a connection, such as a BX.25 session, require a significant amount of

* A transmitter FTS is a program that establishes connection with a receiver FTS to transfer a sequence of messages. A transmitter FTS can be either the initiator or the source. Similarly, the receiver FTS can be either the source or the destination.



EOH – END OF HEADER

Fig. 2—Format of a peer-level message.

network resources, which is not justified for short message exchanges. A Connectionless service must provide the following service primitives:

1. *Send* (*Destination Address*, *message*). This primitive is used to send a block of data called *message* to the FTS in the computer with the address *Destination Address*.

2. *Receive* (*message*). The local FTS uses this primitive to receive a block of data called *message* from the data transport entity.

The data transport services described here are used to transport the peer-level messages described in the next section.

V. PEER-LEVEL PROTOCOL

File transfer systems resident in different computers interact with each other using the peer-level protocol described in this section. The peer-level protocol consists of the procedures for message exchange and message formats. The messages are transported using the data transport services described in the previous section.

Since the peer-level protocol gives the rules for interaction between any pair of FTSs, it is the core of the global coordination functions. The rules and the formats specified in this section must be uniformly implemented in *all* FTSs.

5.1 Message formats

All fields in the peer-level messages have been encoded using the International Organization for Standardization (ISO) communication heading format standard.[4,8] Figure 2 shows a general message format. The first field, the message type field, identifies the message type; it is one byte long. Encodings for various message types are given in Table I. The first field is followed by one or more parameter fields, each of which carries a parameter, such as a file descriptor. The first byte uniquely identifies the parameter type, such as file descriptor. Codes for various types of parameters are given in Ref. 2. Each parameter field can be of variable length, with a maximum of 255 bytes. The list of parameters is followed by an end of heading field, H'80'.* The last field contains data for a file data message.

* H stands for hexadecimal. H 'xx' is used to represent a two-digit hexadecimal number.

Table I—Message type codes

Message Type	Message Name	Message Type Code
Command	Authenticated COPY command, CAC	H'24'
	Negotiation package, CNEG	H'31'
	Checkpoint command, CCM	H'32'
	Inquiry about checkpoint command, CIC	H'33'
	Positive confirmation message, CPC	H'34'
	Negative confirmation message, CNC	H'35'
	Inquiry command, CIO	H'36'
	Cancel command, CCC	H'37'
	Interrupt command, CIN	H'38'
	Restart command, CRS	H'39'
Responses	Acknowledgment, RAC	H'41'
	Reply negotiation package, RRN	H'42'
	Checkpoint response, RCH	H'43'
	Status Response, RSR	H'44'
	Cancel Response, RCR	H'45'
	Notification message, RNM	H'46'
Data Messages	File Header Message, DHM	H'01'
	File Data Message, DDM	H'03'

5.2 Rules for message exchanges

The rules for message exchanges for the Copy and Status services are given in Sections 5.2.1 and 5.2.2, respectively. The rules for the Cancel service are similar to those for the Status service and are not given here.

5.2.1 Copy service

For the Copy service, FTSs in the initiator, the source, and the destination go through the following phases:

- Transfer of an authenticated copy command from the initiator FTS to the source FTS (if the initiator FTS is not the same as the source FTS),
- Negotiation phase,
- File transfer phase,
- Notification phase.

After the user issues a copy command, the initiator FTS checks the access privileges of the user initiating the file transfer and the syntactic correctness of the copy command. If the user does not have the

necessary access privileges and the syntax of the copy command is incorrect, the initiator FTS returns an error message to the user and the file transfer is abandoned. Otherwise, the initiator ships the copy command to the source only if the initiator is different from the source in a peer-level message called the Authenticated Copy Command (CAC). After sending the CAC, the initiator FTS waits for an acknowledgment from the source FTS. If it does not receive an acknowledgment within a time-out period FT1, the initiator will again send the message CAC to the initiator. The initiator makes at most A1 attempts at sending the authenticated copy command.

After receiving an authenticated copy command, the source FTS sends an acknowledgment to the initiator FTS and then enters the negotiation phase. During the negotiation phase, the source and destination FTSs negotiate about options for the file transfer, such as selection of pre- and postprocessors, the values of the intercheckpoint interval and the window size for checkpointing. Intercheckpoint interval and window size are defined later in this section. Further details about the negotiation phase are given in Section VI.

If the negotiation phase is successful, the source and destination FTSs enter the file transfer phase. Each file is transferred completely before the transfer of the next file is initiated. The source FTS sends the following sequence of peer-level messages to the destination FTS for each file transmitted:

- File header message: the file header message carries the name of the file in which the file to be transferred will be stored, the file's attributes, such as code set type and file length,
- One or more file data messages (a file data message is a peer-level message that contains a part of the file contents),
- One or more checkpoint commands (a checkpoint command is a peer-level message which marks a checkpoint during a file transfer).

Checkpoint commands are sent between file data messages at every intercheckpoint interval in the file contents. The intercheckpoint interval is the amount of file data sent between two checkpoints. After sending the checkpoint command, the source FTS starts a timer with the time-out period FT3. It keeps on sending additional file data until the number of outstanding checkpoint commands is equal to the window size for checkpointing, defined as the maximum allowed number of outstanding commands. On receiving a checkpoint command, the destination sends an acknowledgment called the checkpoint response to the source.

The checkpoint commands are assigned sequence numbers. Since the sequence numbers cannot be infinitely large, a cyclically reusable sequence numbering scheme is used. We take the sequence range to

be 0 to $(2^{32} - 1)$, where 2^{32} is the size of the sequence space. The number of outstanding checkpoint commands is limited to w, the window size for checkpointing. The checkpoint command and the checkpoint response carry the sequence number of the checkpoint. The source's reception of a checkpoint response with the sequence number x means that the destination FTS has received everything correctly up to the checkpoint assigned the sequence number x.

End of File (EOF) is marked by sending a checkpoint command. The source FTS must receive the checkpoint response to the checkpoint command indicating EOF before it initiates the transfer of the next file. The checkpoint command indicating EOF for the last element in the file transfer request must also indicate whether additional file data will be transferred during the current session.

5.2.2 Status service

Now, we give the rules for the Status service. A user can issue a status command at the initiator FTS to inquire about the status of a file transfer identified by a job number. The initiator FTS checks if it has sent out the authenticated copy command corresponding to the job number given in the status command. If it has not, the initiator FTS informs the user that preprocessing, file transfer, and postprocessing have not started. Otherwise, the initiator FTS sends an inquiry command to the source FTS. The source FTS checks to see if it has completed preprocessing and file transfer. If not, it sends a status message to the initiator FTS, stating the steps it has already taken and the step presently in progress. If the source FTS has already completed preprocessing and file transfer, then it sends an inquiry command to the destination FTS. On receiving the inquiry command, the destination FTS sends a status message indicating whether postprocessing has been completed successfully or unsuccessfully. The source FTS combines this information from the destination FTS with the status information it has about preprocessing and file transfer. The source FTS sends a status message to the initiator FTS. The status message contains the following information:

- Which stages of the copy command (preprocessing, file transfer, and postprocessing) have been completed,
- The nature of the error if one of the stages described above ended in error.

The initiator FTS reports the status information received to the user.

VI. NEGOTIATION PHASE

A file transfer using the FTP has available a choice of several options. For example, checkpointing may or may not be used during

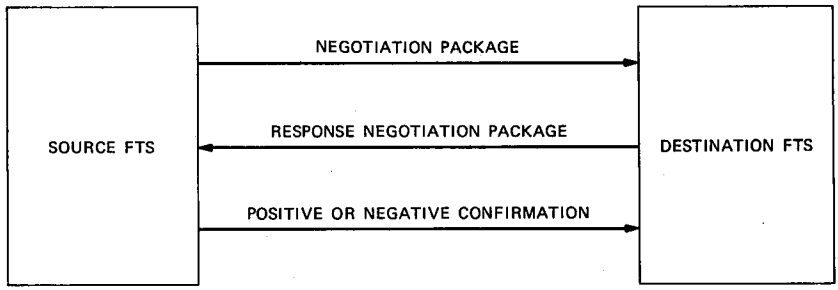


Fig. 3—Message exchange during the negotiation phase.

file transfer; if checkpointing is used during a file transfer, then several parameters, such as the intercheckpoint interval and the window size, must be agreed upon between the source and the destination. These parameters will determine the buffer requirements during the file transfer at the source and the destination. Another parameter to be chosen is the list of postprocessors to be used at the destination.

Before the file transfer phase starts, the source FTS and the destination FTS must negotiate regarding the parameters to be used during the file transfer and come to an agreement. For this negotiation, a three-way message exchange between the source FTS and the destination FTS is used, as shown in Fig. 3.

In the first step, the source FTS sends a negotiation package to the destination and then waits for a reply negotiation package. The negotiation package contains the following information about the parameters to be used during the file transfer:

- Block size. This field specifies the size of the file data transported in one file data message.
- List of postprocessors.
- Whether or not checkpointing is to be used.
- Intercheckpoint interval. This field specifies in units of bytes the amount of file data transmitted between two checkpoints. This field is required only if the checkpointing option is chosen.
- Window size for checkpointing. This field contains the maximum number of outstanding checkpoint intervals at the source FTS. If this field is blank, then the default value is 1.

In the second step, the destination FTS examines the options for file transfer after receiving a negotiation package from the source FTS. The destination FTS decides whether it can accept them. If the destination FTS cannot accept some or all options, it selects alternatives that it can accept. It puts acceptances, rejections, and alternatives in a response negotiation package and sends them to the source FTS. A reply negotiation package contains the following information:

- Specification of rejection or acceptance of each option identified in the negotiation package,
- Suggested alternatives to the rejected options.

In the third step, on receiving a reply negotiation package, the source FTS replies with a positive confirmation message if the destination FTS accepted all options or if the source FTS can accept the alternatives given in the reply negotiation package. Otherwise, the source FTS will send a negative confirmation message to the destination FTS.

If the source FTS does not receive a reply negotiation package from the destination FTS within a time-our period FT2 after sending a negotiation package, it sends the negotiation package again. If the source FTS does not succeed within A2 attempts, it informs the initiator FTS about the failure and aborts the file transfer.

One parameter that the source and destination FTSs might negotiate on is the time when the file transfer should be started. The FTP described here does not negotiate on the starting time of the file transfer, but it can be easily extended to do so. If the underlying network only carries file transfer traffic, then such negotiation can be very useful.[9] Otherwise, we feel that it should not be used for the following reason. Such negotiation requires that the FTSs should be allowed to schedule the future allocation of connections, but this is not consistent with the OSI reference model approach.[5] Several application layer protocols must share the same network resources. Scheduling of the connection allocation, if any, must be done by the lower layers, not by an application layer protocol, such as the FTP.

For networks that carry file transfers exclusively, negotiation about the starting time can be very useful. File transfers typically require a heavy commitment from the network. A connection used for a file transfer typically utilizes nearly 100 percent of its maximum transfer rate as defined by its throughput class. On the other hand, a connection carrying interactive data utilizes only 1 to 2 percent of its maximum transfer rate. As a result, the number of outgoing and incoming connections used for file transfers is very small. Scheduling of connections and file transfers to optimize a cost function, such as network utilization, or minimizing delay can be very useful.

VII. EXAMPLES

We give two examples to illustrate the use of the FTP: one for the Copy service and one for the Status service.

In the first example (Fig. 4), a user issues a copy command to the file transfer system in Computer-C to copy a file FSOU from Computer-A to the file FDEST in Computer-B. The file transfer system in Computer-C (henceforth, referred to as FTS-C) checks the com-

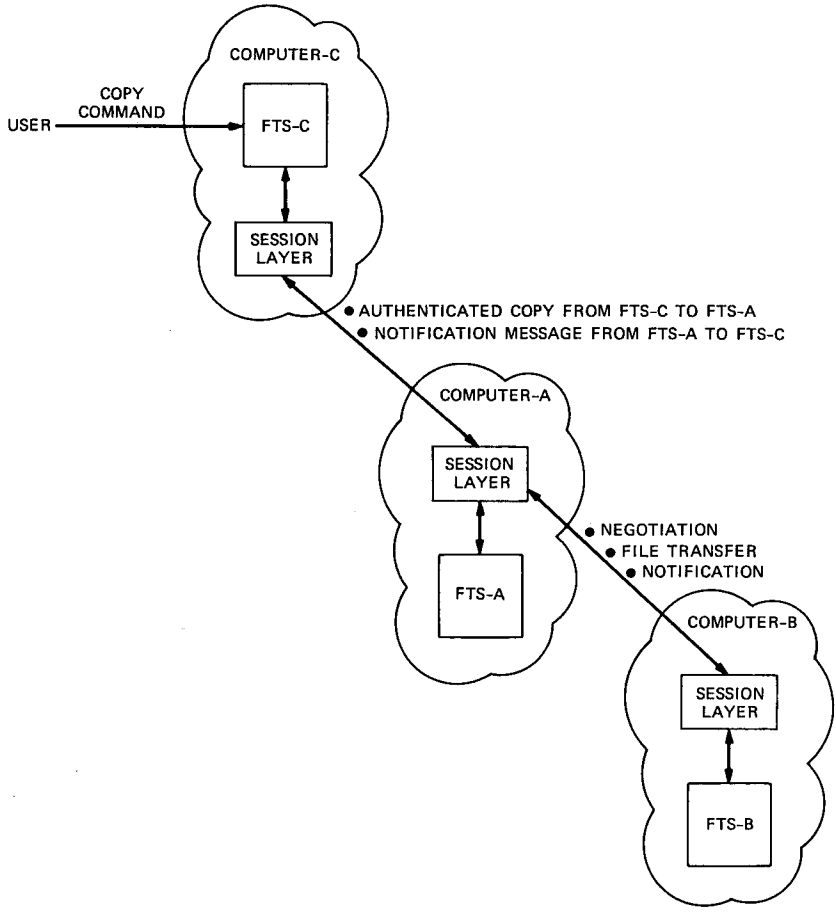


Fig. 4—An example of the Copy service.

mand for its syntactic correctness and checks the access privileges of the user to find out whether the user is allowed to initiate this file transfer.

If the copy command has a syntax error, then FTS-C returns an error message to the user. If the user has the proper access privileges, then FTS-C returns a job number to the user. The job number is a unique identifier assigned to the copy command. It uniquely identifies the command throughout the network. The format for the job number is given earlier in Section III.

FTS-C sends an authenticated copy command to the FTS in Computer-A (henceforth, referred to as FTS-A). Then, it waits for an acknowledgment from FTS-A. If FTS-C does not receive an acknowledgment within the time-out period FT1, then it retransmits the

authenticated copy command. It makes at most A1 transmission attempts.

On receiving the authenticated copy command from FTS-C, FTS-A negotiates with FTS-B for the options of file transfer such as choice of pre- and postprocessors, the checkpointing option, and the parameters for checkpointing. If the negotiation is successful, then the actions described below take place. Otherwise, FTS-A sends an error message to FTS-C and no further operation takes place.

FTS-A sends a file header message to FTS-B. A file header message contains the information such as the file length and the file name. Then, FTS-A processes the file data using the preprocessors and divides it into blocks. Each block is packed into a separate file data message. Then, FTS-A sends these file data messages to FTS-B. It also sends the checkpoint commands at every intercheckpoint interval. If the session being used for the file transfer breaks down, then FTS-A can resume file transfer from an intermediate checkpoint up to which FTS-B has received the data correctly. FTS-B unpacks the file data in the messages received. Then, it processes the file data received using the postprocessors agreed upon during the negotiation phase. FTS-B then stores the file data in the file FDEST.

After the successful file transfer, FTS-B sends a notification message to FTS-A informing it about the successful completion. FTS-A, in turn, sends a notification message to FTS-C informing it about the successful completion. Finally, FTS-C informs the user about the successful completion of the file transfer.

In the second example, we illustrate use of the Status service (Fig. 5). A user issues a status command at the local FTS, FTS-C, to inquire about the status of the copy command described above. The status of a copy command consists of at least the following information:

- Whether preprocessing has begun. If yes, whether it has been completed successfully.
- Similar information about file transfer and postprocessing. The status can further include additional information, such as how far the file transfer has progressed.

FTS-C checks the status command for its syntactic correctness. If the command has a syntax error, then FTS-C returns an error message to the user. Then, FTS-C checks the access privileges of the user to determine whether the user can inquire about the file transfer. If the user has the necessary privileges, then FTS-C checks the local records to find out whether the file transfer has been completed. Otherwise, FTS-C sends a status command to the destination FTS, FTS-B. FTS-B checks the local records to locate the status of the file transfer. It checks whether it was ever involved in that file transfer. If it was involved in the file transfer, FTS-B checks whether the file transfer

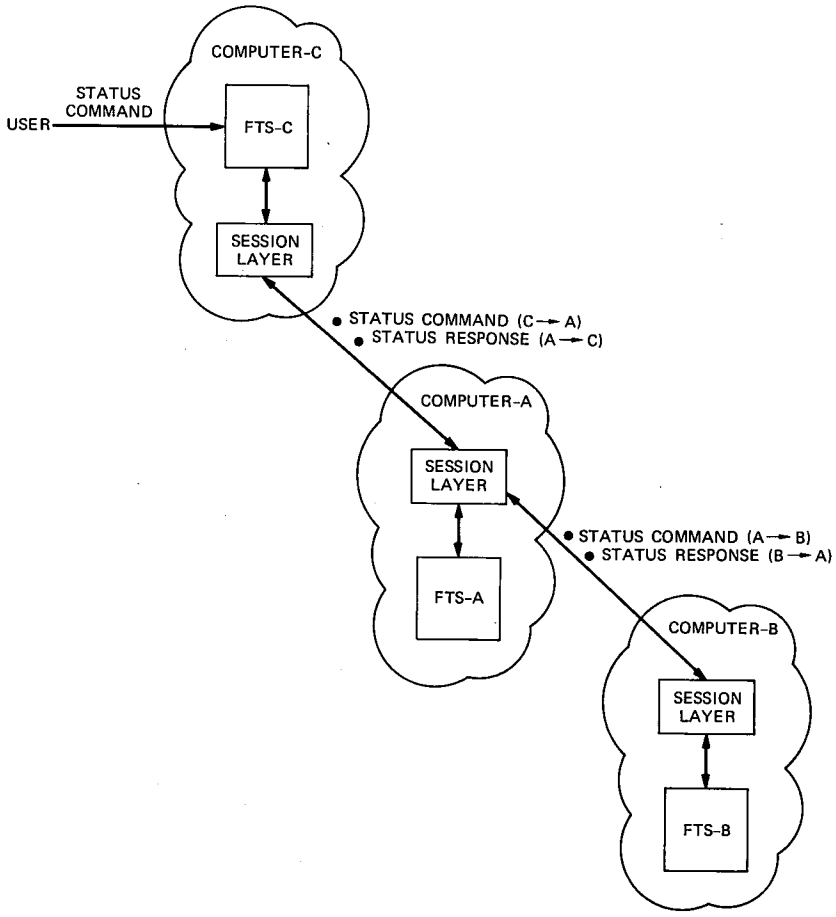


Fig. 5—An example of the Status service.

has been completed or if it is still in progress. FTS-B sends the status information in a status response message to FTS-A. FTS-A also collects the local information about the file transfer and combines it with the information received from the destination FTS, FTS-B. Then, it sends the status information to FTS-C. Finally, FTS-C delivers the status information to the user.

VIII. FORMAL SPECIFICATION OF THE FTP

We have specified the FTP[8] in a mathematically precise notation based on the *selection/resolution* (s/r) model of coordination described in Refs. 6, 7, 10, and 11. Compared with the English language specification, the formal specification provides a more precise and complete

description of the protocol. In this section, we first present an informal discussion of the s/r model; the reader is referred to the references for more details. We then discuss how the FTP was formally specified using this approach.

8.1 The selection/resolution model

The selection/resolution model is a method in which a complex system such as a protocol can be represented as a set of coordinating finite state machines. Each component finite state machine, called a *process*, is described by an appropriately labeled directed graph. For instance, suppose we wished to describe the coordination of two processes A and B that can be executing segments of code in either a noncritical section or a critical section. We wish the coordination to be such that both processes are not simultaneously in their critical sections. Figure 6 has two labeled graphs representing the processes.

In the labeled graphs of Fig. 6, the vertices represent *states* of the process, and the edges represent possible single-step transitions. Thus, process A (also B) can be in the states NCS (Noncritical Section), TRY (trying to enter the critical section), and CS (Critical Section). A state encapsulates the relevant past of the process needed to determine future behavior and is known only to that process. The processes coordinate by exchanging information about their future intentions. That is, they communicate their *selections* (intentions) to all the other processes. For example, in state NCS, process A can only choose the selection *ok*, while in state TRY it can nondeterministically choose between *head* and *tail*. Note that a process can make only one selection at any time. Selections of a process are given in curly braces next to the state.

Each edge has a label next to it. The edge labels are Boolean conditions based on the selections of all the processes, and determine the possible transitions of a process. In conditions, + is the same as a logical *or* and * is the same as a logical *and*. For instance, the condition for process A to go from CS to NCS is simply (*A:nok*), that is, it really doesn't depend on B, and is in fact always true since the only selection of A in state CS is *nok*. However, the condition for A to enter the CS state from the state TRY is [(*B:ok*) + (*A:head*)*(*B:head*) + (*A:tail*)*(*B:tail*)]. In simple words, the condition says that A can go to the CS state if B selects *ok*, or if A and B both select *head*, or if A and B both select *tail*. The transition of process A from state TRY to its next state clearly depends on what B is selecting. Both processes essentially toss coins to determine who wins if they are both trying to enter the critical section. The transition of a process to one of the possible next states (there may be more than one transition enabled) is called *resolution*.

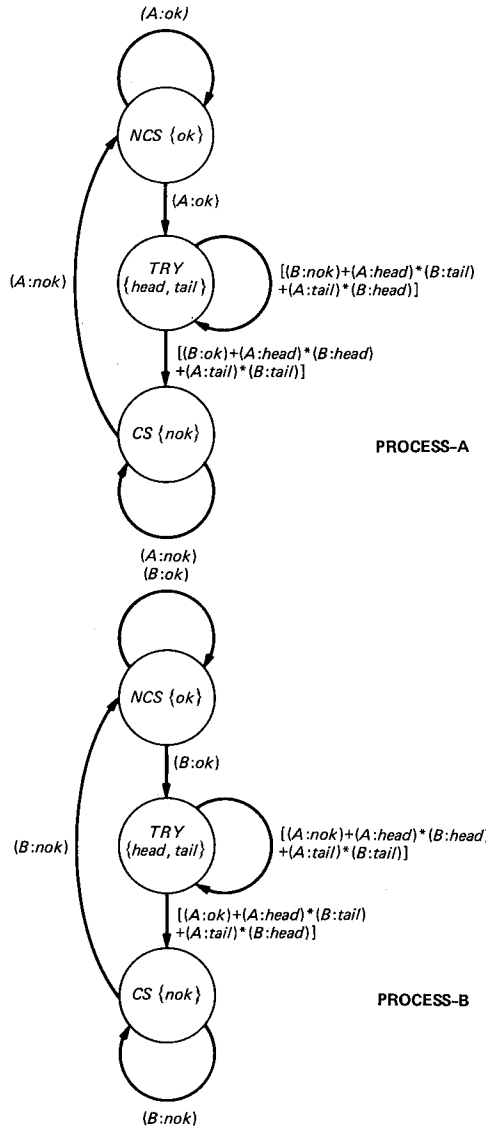


Fig. 6—Mutual exclusion example.

In the above example, developed in more detail in Ref. 11, both processes initially start in the state NCS at time-step 0. Then, in each unit of time, the processes first make a selection based on the state they are in, and then they resolve (transition to a new state) based on the selection of all the processes. It can be seen that the processes will never both be in the states CS at any time step.

The s/r model gives precise algebraic descriptions of processes that can be combined to give a precise description of the entire system. Furthermore, many computational aspects of this model can be automated. In fact, the formal specification can be used to test the correctness of the FTP by conducting a reachability analysis using the procedure given in Refs. 6 and 7.

8.2 Specification of the FTP

The formal specification of the FTP consists of descriptions of 45 processes divided for convenience into 9 clusters (see Fig. 7). A cluster

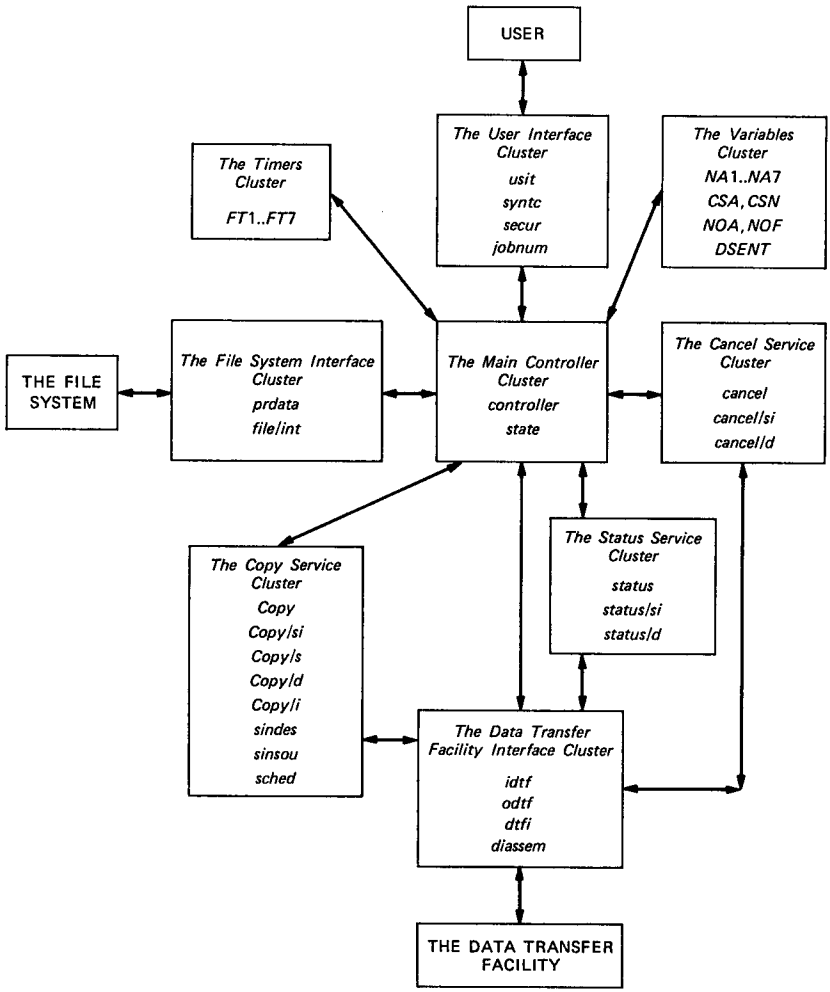


Fig. 7—Block diagram of the formal specification.

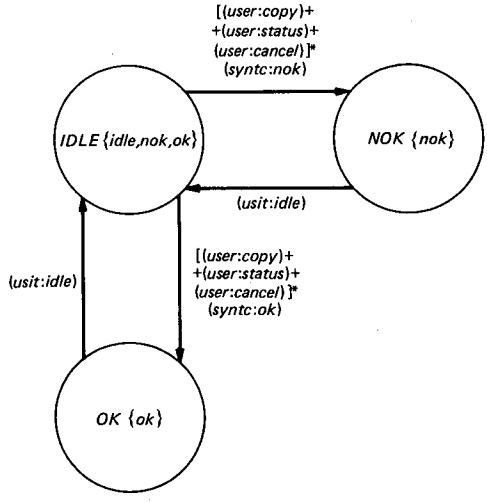


Fig. 8—The Process *syntc*.

is a convenient grouping of a set of processes. We have grouped processes into clusters based on their functions. Thus, there are clusters that correspond to the basic service functions such as Copy service, Status service, and Cancel service. There are also clusters that correspond to interface functions such as interfacing with the user, the data transfer facility, and the file system. Finally, there is a main controller cluster that acts as an overall coordinator, and there are clusters for the timers and the variables.

Each process is defined in a language that essentially provides the same information as the labeled graph corresponding to that process. For example, we can define a process that does a syntax check on the user command. We model this as the process *syntc* shown in Fig. 8. As discussed in Section III, the exact choice of the command syntax is a local decision. Notice that we model only the portion of the process behavior that describes global coordination with other processes. The process *syntc* makes the selection *ok*, only if the command contains all the required information and is syntactically correct. Otherwise, it selects *nok*.

In this figure we have not indicated the conditions for the self-loops.* We shall assume that the self-loop condition is like an *otherwise*; that is, it is the *negation* of the *or* of the conditions on the other edges. The process *syntc* is defined using the specification language in Fig. 9. Notice the close correspondence with the graphical specification. We use $ as shorthand to signify the current state. In this

* A self-loop is a transition from a state back to the same state.

```
    process     syntc/* A process to do syntax check of the user commands */

selvar: valr 0..2
       valnm    idle, ok, nok

stvar: valr 0..2
      valnm    IDLE, OK, NOK

initial condition: IDLE/(syntc:idle)

trans:

IDLE                {idle, nok, ok}

  > OK              :((user:copy) + (user:status) +
                       (user:cancel)) * (syntc:ok))
  > NOK             :((user:copy) + (user:status) +
                      (user:cancel))*(syntc:nok))
  > $                 :;

OK                 {ok}

  > IDLE            :(usit:idle)
  > $                  :;

NOK                {nok}

  > IDLE            :(usit:idle)
  > $                 :;
```

Fig. 9—Specification of Process *syntc*.

example, the empty condition on the self-loop is equivalent to *otherwise*.

Process *syntc* is one of the processes in the User Interface Cluster. This cluster consists of four processes: *usit*, *syntc*, *secur*, and *jobnum*. Each process's description is on the average about as complex as that of *syntc*. The process *usit* gets into one of the three states—COPY, CANCEL, and STATUS—if the user command is syntactically correct and if the user has the permission to execute the command. The process *secur* is similar to *syntc*; it checks the security privileges of the user. It makes the selection (*secur:ok*) if the user can execute the command. Otherwise it makes the selection (*secur:nok*). The process *jobnum* generates a job number for each copy command. The job number uniquely identifies a copy command throughout the network.

For the processes *secur*, *syntc*, and *jobnum*, we model only their synchronization and interaction with other processes. An FTS developer must design the appropriate procedures used in these processes. The interaction, however, should be exactly as described. The English language specification of our FTP does not define what procedures to use to do syntax checking or security checking, etc. Thus, the formal specification must not define the procedures to be used either.

The full formal specification (about 50 pages long) consists of similar descriptions of all the clusters. The descriptions of the processes, of course, vary in complexity. The formal specification can be used as a precise guideline by FTP implementors. In fact, the high-level design

of the FTP implementation may be readily derived from the formal description. A development group at AT&T Communications used the formal specification in their design process. They found it to be extremely useful in developing a high-level C-like language design of FTP.

For implementation, each user command can be viewed as invoking an independent copy of the 45 processes. If an FTS must handle several file transfer commands at the same time, then an independent copy of the processes is invoked for each command. Since our FTP is a three-party protocol, a file transfer at an FTS can be initiated from a remote FTS. For a file transfer initiated from a remote FTS, an independent copy of the processes is invoked at the local FTS. The FTS can thus be implemented as a reentrant program that can be used independently by each command.

IX. COMPARISONS WITH OTHER FILE TRANSFER PROTOCOLS

As discussed previously, our approach here is to separate the file transfer issues from local service functions. This allows great flexibility in local matters and avoids proscribing rigid formats for the user interface, for file management and naming, and for presentation functions such as code sets. Our FTP differs in this regard from other file transfer protocols that generally are rather inflexible and less adaptable. For example, none of the other file transfer protocols has a general mechanism such as pre- and post-processors that allow additional capabilities to be incorporated in a standard way. In this section, we compare the FTP with four file-transfer protocols: Arpanet FTP,[12] Network Independent (NI) FTP,[13-15] Autodin II FTP,[16] and the International Organization for Standardization (ISO) File Service Protocol.[17]

The Arpanet FTP requires the user (initiator) to establish separate command and data connections with both the source and the destination of the file transfer. The user must keep track of these connections. The user is also involved at too detailed a level with the file transfer operation; it is not possible to simply copy a set of files from the source to the destination with a single macro command. It is expected that the user remain on-line at each phase of the transfer, and the user must be aware of details such as the socket number of the data connection. The protocol does provide for both copy and append services, as well as status notification. File management services such as creating and deleting files are also available. Security measures are fixed and encompass only user name, password, and account number. There is no provision for adding other local security options.

The NI FTP is a two-party protocol (no provision is made for third party initiation of a file transfer) that divides the transfer operation into three phases. The protocol handles only transfers of single sequential files; multiple files must be sent as separate file transfers. The NI FTP protocol provides for a high degree of flexibility in the actual data transfer operation. This is accomplished through the negotiation of a large number of possible alternatives. The possible alternatives are defined in any particular implementation, and cannot be readily modified. There is a very limited file management capability in NI FTP. The protocol assumes only minimal support from the lower layers, and specifies presentation services. It allows for modular implementation, but does not have provisions for easily adapting to new requirements. The protocol provides for only foreground file transfers.

The Autodin II FTP provides high-level service primitives and allows background file transfers. However, it goes too deeply into defining the structure of a file at the network level; this results in the negotiation of a large number of parameters for the actual data transfer, rather than allowing this to be handled through local processing. Furthermore, each of the three parties involved in the file transfer must keep an elaborate list of parameters that define all the options that have been agreed upon. There is an extensive security mechanism, but it is not modifiable. Similarly, the user interface is very specific, and cannot be adjusted to local preferences. Autodin II FTP is definitely comprehensive but all the detail is defined at the network level. This results in a protocol that requires a complex implementation at each node.

A committee of the ISO is working on the File Access, File Transfer, and File Management (FTAM) services and on a corresponding protocol. The FTAM services allow a user to access and transfer a remote file. They also allow a user to manipulate a file in a remote file system. An example of a service is a command to open a file in a remote file system. These are micro services. In contrast, our FTP provides macro services, such as the copy service. Each service of our FTP can be provided by a long sequence of the ISO service calls. We have carried out a study to define these sequences of the ISO service calls.[18]

The ISO file service uses the concept of the Virtual File Store. The Virtual File Store is a common model for describing file names and their attributes. Different file systems have a wide range of styles for describing the storage of data and the means by which it can be accessed. The Virtual File Store allows the differences in style and specification to be absorbed in a local mapping function. The Virtual File Store attempts to encompass all possible variations of file defi-

nition, resulting in an excessively detailed description of a file that cannot be easily modified in the future.

X. CONCLUSIONS

In the FTP presented here, we separated global functions requiring close coordination (such as parameter negotiation) from functions that could be done locally at each individual node. In an FTP implementation, the global functions *must* be implemented. Local functions can be implemented depending on user needs. As a result, a minimal implementation of the protocol is fairly simple. Furthermore, it requires only incremental efforts to extend the minimal implementation.

In our approach, we first defined the service specification and the services required by the protocol from the lower layer. Then, we designed the peer-level protocol that fills the gap between the upper and lower layers. Adding service specification leads to a clearer specification of the entire protocol and is useful to implementors. We have also described a formal specification of the protocol that can be used to resolve any ambiguities in the English-language document.

We feel that the FTP is flexible, adaptable, and powerful enough to meet most file transfer requirements. Furthermore, it has been specified precisely enough so as to make implementation a relatively straightforward task.

REFERENCES

1. M. E. Grzelakowski, J. H. Campbell, and M. R. Dubman, "DMERT Operating System," B.S.T.J. *62*, No. 1, Part 2 (January 1983), pp. 303–22.
2. *Operations Systems Network Protocol Specification: BX.25 Issue 3 Addendum-A*, AT&T Bell Laboratories, Holmdel, NJ, August 1983.
3. H. R. Patel and G. S. Lohr, unpublished work.
4. *Operations Systems Network Protocol Specification: BX.25 Issue 3*, AT&T Bell Laboratories, Holmdel, NJ, June 1982.
5. A. S. Tanenbaum, *Computer Networks*, Englewood Cliffs, N.J.: Prentice-Hall, 1981, Chapter 1, pages 10–21.
6. R. P. Kurshan and B. Gopinath, Unpublished work.
7. S. Aggarwal, R. P. Kurshan, and K. Sabnani, "A Calculus for Protocol Specification and Validation." In *Protocol Specification, Testing, and Verification, III*, edited by H. Rudin and C. H. West, Amsterdam: North-Holland, 1983.
8. S. Aggawal and K. Sabnani, Unpublished work.
9. E. G. Coffman, Jr. et al., "Scheduling File Transfers in a Distributed Network," Second Annual Symp. on Principles of Distributed Computing, 1983, pp 254–6.
10. S. Aggarwal, R. P. Kurshan, and D. K. Sharma, "A Language for the Specification and Analysis of Protocols," In *Protocol Specification, Testing, and Verification, III*, edited by H. Rudin and C. H. West, Amsterdam: North-Holland, 1983.
11. S. Aggarwal and C. Courcoubetis, "Distributed Implementation of a Model of Communication and Computation," Proc. 18th Hawaii Int. Conf. on System Sciences, January 1985, pp 206–218.
12. N. J. Neigus, "File Transfer Protocol for the ARPA Network." In *ARPANET Protocol Handbook*, edited by E. Feinler and J. Postel, Defense Communications Agency, 1978.
13. C. J. Bennet and D. N. Frost, "Network Independent File Transfer," Tech. Report, University College, London.
14. R. W. S. Hale, "File Transfer Protocols—Comparison and Critique," NPL Report DNACS 48/81, Teddington, United Kingdom, 1981.

15. High Level Protocols Group, "A Network Independent File Transfer Protocol," HLP/CP(78), NPL, Teddington, United Kingdom, 1977.
16. H. C. Forsdick, "Autodin II File Transfer Protocol," Bolt Beranek and Newman Inc., Report No. 4246, Boston, 1980.
17. *Second Draft Proposal on File Transfer, Access, and Management,* ISO/TC97/SC21 No. 8571, 1985. (Available from ANSI)
18. D. Lewan and K. Sabnani, unpublished work.

AUTHORS

Sudhir Aggarwal, B.S. (Mathematics), 1969, Stanford University; M.S. (Mathematics), 1971, and Ph.D. (Computer and Communication Sciences), 1975, University of Michigan; Assistant Professor of Computer Science, University of Oregon, 1975–1976; Research Scientist, Lawrence Livermore Laboratory, 1976–1977; Assistant and Associate Professor of Mathematics, University of California, Riverside, 1977–1982; AT&T Bell Laboratories, 1982—. Mr. Aggarwal is a member of the Mathematical Sciences Research Center. His research interests are computer communication protocols, local area networks, and modelling and simulation.

B. Gopinath, M.Sc. (Mathematics), 1964, University of Bombay; Ph.D. (E.E.), 1968, Stanford University; Research Associate, Stanford University, 1967–1968; Alexander von Humboldt Research Fellow, University of Gottingen, 1971–1972; Gordon McKay Professor, University of California, Berkeley, 1980–1981; AT&T Bell Laboratories, 1968–1983; Bell Communications Research, 1983—. Mr. Gopinath is Division Manager for Systems Principles Research, and is engaged in research in communications and computer science.

Krishan Sabnani, B. Tech. (E.E.), Indian Institute of Technology, New Delhi; Ph.D. (E.E.), Columbia University, New York, 1982; fellowship, School of Engineering and Applied Sciences, Columbia University, 1977; research assistant, Columbia University, 1978 to 1979; he was employed by RCA, Princeton, 1979–1981, AT&T Bell Laboratories, 1981—. Mr Sabnani's current interests are computer communication protocols and fault-tolerant computing. Member, Sigma Xi, Eta Kappa Nu, and Epsilon Pi Upsilon.