LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-288

DSCRIBE: A SCRIBE SERVER

Janice C. Chung

October 1985

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

DSCRIBE: A SCRIBE SERVER

Janice C. Chung

May 31, 1985

Abstract

This document gives a complete description of the design and implementation of Dscribe, the Scribe server. Dscribe is a program which allows users on a variety of hosts to have files processed remotely by the Scribe document preparation system. The first part of the document describes the functionality of Dscribe and the motivation for writing the program. It also gives an overview of how the program works. Later sections discuss important design issues and describe the implementation in detail.

KEYWORDS: network services, Scribe, document formatting

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Number N00014-83-K-0125.

Table of Contents

1. Introduction

1.1. Functionality

"Dscribe" stands for "Distributed Scribe". The primary function of Dscribe, the Scribe server, is to make the Scribe document preparation system available to Laboratory members working on a variety of host computers. Dscribe's other functions are to print files on the Dover or Imagen printers and to pressmerge press files together.

The Scribe document preparation system consists of the Scribe language and the Scribe compiler. The user specifies the desired format of his document by inserting high-level *commands* from the Scribe language into his text file; the combination of text and commands is called the *manuscript file*. The Scribe compiler processes the commands in the manuscript file and generates the *finished document*, a file which can be output on a certain printing device to yield a fully formatted document.

The Scribe program is written as a single-machine application. It runs on a particular machine and serves only that machine's users. Dscribe turns this single-machine application into a service, allowing it to be used by a community of users on a network. A user can edit files locally, then call Dscribe to have those files remotely Scribed. Advantages of this "server approach" are discussed in the following section.

Dscribe's two other functions, pressmerging and printing, provide additional services that are often needed in conjunction with Scribing a file. A *press* file is a file that has been specially formatted to be output on a Dover printer. *Pressmerging* combines a main press file with one or more one-page press files and is useful for inserting figures into a document. The printing function prints a *finished document* generated by Scribe on the appropriate printing device. Currently, the accessible printers are the Dover printer on the ninth floor and the Imagen printer on the fifth floor.

The initial design for the Scribe server included two additional functions which have not been implemented: (1) checking the status of earlier Scribe requests, and (2) maintaining the Scribe database. According to the original design of Dscribe, a user could initiate a Scribe process on a manuscript file, then "disconnect" from Dscribe while Scribe continued its processing. The status-checking function allowed the user to check, at some later time, the status of the Scribe run from which he disconnected and to receive any output files at that time. The current implementation of Dscribe does not allow the user to disconnect in this manner; hence, the status-checking function is unnecessary.

Dscribe's original design prohibited the use of private databases, and instead proposed support for maintenance of the standard Scribe database so that users could alter the standard database used by the server. Dscribe's design has since been modified to accommodate the use of private databases; users may specify a private database directory via a Scribe command in the manuscript file. With this change, the user no longer needs to be able to change the standard database. Consequently, the database maintenance function has not been and will not be implemented.

1.2. Why have a server?

The principal motivation for writing Dscribe was to broaden the community of users that could use the Scribe document preparation system. One group of users who have gained access to Scribe consists of the IBM PC users in the Laboratory. Before Dscribe existed, PC users had to transfer their files to another computer which did support Scribe, log on to that computer, and run Scribe there. Now these users can edit files locally, then run Dscribe *on the PC* to have their files remotely Scribed.

There are several other reasons for implementing an application (not necessarily Scribe) as a service rather than as a single-machine program. Before explaining them, however, we need to introduce some terminology. A service program is divided into two parts: the *back-end* and the *front-end*. The *back-end program*, or the *server program*, is the part that manages the application and provides its services to users; the machine that it runs on is called the *back-end machine*. The *front-end program*, or *client program*, is the part that runs on the user's host machine, also known as the *front-end machine*. The user invokes the client program, and the client program communicates with the server program to request service on the user's behalf.

Now we list some reasons for writing an application service:

1. As mentioned above, by calling on a remote application server, a user can access a resource that for some reason is not available on his own host machine (e.g. inadequate space).

2. Even if the application does run on his host machine, the user might want to call a remote server in order to reduce the load on his own machine.

3. When an application program is designed as a server, it is easy to make the service available to users on different kinds of hosts. Instead of having to write the application program to suit each particular system, one need only write one server program for the back-end machine, and one client program for each type of front-end machine. The client should be designed to be small and easily portable.

4. Since the client program is generally much smaller than the application program, the server approach saves space on the front-end machines. (This may be important for small machines.)

5. The application program is easier to update since, in many cases, only the program on the back-end machine needs to be modified.

6. Use of an application server produces consistent output no matter what host machine the user is on, whereas the output from machine-specific application programs may vary slightly from machine to machine.

1.3. User interface

Dscribe was designed to be fairly transparent to its users. Running Dscribe is very similar to running Scribe. Files do not have to be altered in any way to accommodate the server; any file that can be processed by Scribe can also be handled by Dscribe. Also, users need not worry about transferring files to another machine; this is automatically handled by the server.

When a user types "dscribe" at his host machine, he is presented with a menu and asked to select the desired function: *Scribe*, *Print*, *Pressmerge*, *Help*, or *Quit*. By supplying a filename and certain option flags on the command line, the user can bypass the menu; the proper syntax for such a command line is described below for each function.

If the *Scribe* function is selected, the user is asked for the name of the manuscript file. The user can bypass the menu with a command line of the following form: dscribe <manuscript file name> <optional list of Scribe option codes>. The server fetches a copy of the manuscript file and does some preprocessing on it before having it Scribed. Preprocessing involves checking if other files are required and doing the appropriate file transfers. If the server and client programs cannot locate a file, the server prompts the user for a new filename. When Scribe has finished processing the manuscript file, the user is asked (1) whether he wants the finished document transferred back to the front-end, and (2) whether he wants the document printed. However, if the front-end is an IBM PC and the document is a press or imp file, the first choice is not offered. Press and imp files are not sent to PC front-ends because of the limited space on PCs; instead, these files are stored on the back-end. The PC user may still request that the document be printed. All other Scribe output files are automatically shipped to the front-end.

If the *Pressmerge* function is selected, the user is asked for the name of the main press file. The menu can be bypassed with a command line: dscribe <name of main press file> -pressmerge. The Scribe server preprocesses the main press file, then has it pressmerged. Afterward, it asks the user if it should ship the new press file (with filename suffix ".mpress") to the front-end and/or to the printer for output.

The user may also ask the server to *Print* a Scribe document file. To bypass the menu, the command line should be of the form: dscribe <document file name> -print. The user may request printing of a file that he has stored on the front-end machine or one that he believes is stored on the back-end machine. The Scribe server keeps files on the back-end for up to a day, so it can print a document file that was recently generated by Dscribe, even if there is no copy of it on the front-end. The user can refer to such files by names that are meaningful on the front-end machine. The printing device is determined from the suffix of the filename. Currently, the accessible printers are the Dover laser printer on the ninth floor and the Imagen printers on the fifth floor.

If the user selects *Help* from the menu, he is presented with a page of information about the functionality and operation of Dscribe. This information is shown in Appendix I. The menu is presented to the user again at the end of the help information.

The user may quit from the Dscribe program by selecting *Quit* from the menu. He may also quit by typing a 'q' when being prompted for a filename or while the application program is running; this causes an irreversible escape from the program.

When the user is prompted for an entry (e.g. menu selection, filename, yes/no answer), he must supply his response within about a minute after the prompt. If he fails to do so, he receives a timeout message. The server supplies a default answer to any yes/no questions; in all other cases, the program terminates. Thus, if, for some reason, the user does a 'control-C' to exit from the client program, he has a limited amount of time in which to continue the program before it times out.

After the user selects a function, he is informed of the jobID of his Dscribe job. This jobID number is to be used for status checking. Since that function is not yet implemented, the user does not need to remember this number.

Dscribe's client program may be run in foreground or background mode, as long as this capacity is provided by the front-end machine.

1.4. Overview of the implementation

The back-end program for Dscribe resides on the MIT-OA Vax and is started up every time the system reboots. The front-end program runs on Vax 11/750 computers, MIT-XX, and IBM PCs. When a user invokes Dscribe, his host machine opens a connection to the back-end machine (MIT-OA). The server, which has been listening for connections, then forks off a *handler* process to handle the requests of this particular user and loops back to wait for the next connection. A total of eight

handlers may be running simultaneously. (The number is limited by the number of pipes available.) If a ninth user tries to connect to the back-end, he is informed that the system is fully loaded and is asked to try again later. A handler process will timeout and terminate if the user fails to answer a question in about a minute; this prevents any one user from tying up a handler for an indefinite period of time.

Suppose the user has selected the Scribe function. The handler communicates with the front-end program across the network and attempts to assemble on the back-end computer all the files needed for the Scribe run. It does this by transferring the user's manuscript file to the back-end and then preprocessing it. This means it scans the file to find implicit and explicit references to other files, then it transfers these files to the back-end, and, if necessary, preprocesses them in turn. (When we speak of transferring a file from the front-end to the back-end, we actually mean sending a copy of it to the back-end.) The intent of the preprocessing is to prevent Scribe from ever having to stop and wait for input once it starts running.

Files transferred from the front-end to the back-end are cached under new names in the server's working directory on the back-end. When Dscribe needs to access a file, it checks to see whether the file exists on the front-end and on the back-end, then compares their timestamps to see which version is more recent. If the front-end has a newer version, it is shipped to the back-end, where it replaces the old version.

When all necessary files are assembled in back-end storage, the handler forks a Scribe process to compile the back-end's copy of the manuscript file. The handler intercepts the output that Scribe would normally display on the user's terminal. Before forwarding it to the front-end program, it replaces any names of back-end files with corresponding front-end filenames so the user will see filenames that he recognizes and expects to see. The front-end program displays the forwarded output to the user. When Scribe finishes, the server transfers the output files to the front-end.

Whatever function the user selects, be it Scribe, Pressmerge, or Print, the back-end program simulates the user and invokes the application program; then it acts as middleman between the application and the real user on the front-end machine. The back-end engages in a constant dialogue with the front-end program, sending it instructions, which the front-end then carries out. In some instances, the front-end routine merely performs the task dictated by the back-end, (e.g. display this message to the user), but usually it also sends a reply in the form of an acknowledgment or information that the back-end requested (e.g. the timestamp of a file or input from the user).

If a user quits from Dscribe or if the front-end program times out, then the handler process associated with that user cleans up and exits gracefully. In order to clean up, it must make any adjustments necessary to reflect the fact that this particular job is finished. For example, the handler must deactivate any files associated with the job. (See section 2.1.)

2. Design issues

2.1. Storage

Manuscript files, auxiliary files, database files, and finished documents produced by Scribe are cached on the back-end machine (as opposed to being deleted after each run) in case they might be needed again by Dscribe at a later time. The directory structure in back-end storage is flat; all files are stored in the server's working directory. This structure was selected instead of a structure allocating one directory per user in order to avoid having unnecessary multiple copies of files on the back-end. The directory is protected so that only the superuser of the back-end machine has access to the files.

The server maintains two tables of information on the files kept in back-end storage. The *name table* maps from the front-end machine name (e.g. MIT-XX, PC) and the file's original name to the server-generated name. When the server copies a file from the front-end to the back-end, it assigns to the file a new name derived from the time that the server forked off a handler process to handle the Dscribe client. The new names are guaranteed to be unique within a one-month time period.

The *file table* maps in the reverse direction, from the server-generated filename to the originating filename. It also includes, for each file, information on the file's timestamp on the front-end, the host name, the user name, the jobID, an "active" flag which indicates whether or not the file is being used in an active Dscribe run, and a "keep" flag which is reserved for use by the status-checking function (not yet implemented).

Dscribe has been designed so that it can handle filenames with or without version numbers. If the front-end machine's operating system does not include version numbers in filenames, the treatment is straightforward. When the timestamp of a front-end file is later than the timestamp of the back-end copy of the same file (or if there is no back-end copy), the front-end file is transferred to back-end storage, replacing any older version there.

If the front-end operating system appends version numbers to filenames, Dscribe treats each different version as a distinct file in back-end storage. (For example, paper.mss.1 and paper.mss.2 are treated as different files in the same way that paper.mss.1 and draft.mss.1 are treated as different files.) Sometimes the user omits a version number, expecting the latest version number to be appended as a default. The Scribe server asks the client program to parse all filenames, so a filename without a version number will be always be given the appropriate default version number.

The information in the name table and file table is used by Dscribe to identify files during preprocessing and to translate between front-end and back-end filenames. As files are transferred to the back-end, new entries are made in the two tables, and the files are "activated" (the active flag is set). If Dscribe discovers that a file it needs is already "active" (being used and possibly altered by another Dscribe run), it simply fetches another copy of the same file from the front-end. This way, two users can simultaneously Scribe the same file. After a run has finished, all relevant files are "deactivated".

The name table and the file table are constantly updated to reflect the current state of back-end storage. Consequently, every handler process must be able to see changes made by the other handlers. Each of the two tables has its contents encoded in a log file in the server's working directory; each handler maintains a local file table and a local name table. When a handler wants to modify a table, it changes its local copy and appends the new information to the table's log file. When it wants to examine a table's contents, it compares the timestamp of its local copy of the table with the time of the last update of the table's log file. If the log file has more recent contents, the handler discards its old table and reads the log file to create a new, up-to-date table. Shared read and exclusive write locks are used to control access to the log files. This scheme allows each handler to have up-to-date information without having to read in a new table for each access.

Back-end storage is cleaned up periodically. Every day at 5 a.m. the back-end process of the server is halted, all files in the server's directory (including the log files for the name table and file table) are deleted, and the server is restarted. If this cleanup did not occur, the log files would continue to grow, and updates to the tables would take longer and longer. The cleanup also prevents the cache from filling up with files that are accessed once and never used again. In the current implementation, files produced by late night runs are not saved overnight. If this approach turns out to be very inconvenient for users, the cleanup program might be changed to delete only those files that have been in back-end storage for longer than some specified amount of time (e.g. 24 hours).

2.2. Communication

Communication between the server and client processes is based on a high-level message passing protocol. Each message contains a header and the text of the message, if any. The header is a numeric opcode that is associated with a particular type of message. The opcode tells the recipient how to interpret the message contents that follow and whether or not a reply is required. Opcodes can be application-specific (e.g. an opcode that tells the front-end "Parse this Scribe JCL into directory, filename, suffix, and Scribe options") or general (e.g. an opcode that says "Display the following text to the user (in whatever way is suited to the front-end terminal) and get an integer

reply"). (See Appendix II for a listing of the headers used by Dscribe.)

Conversation between the back-end and front-end processes is always initiated by the back-end. Once the front-end program has established a connection to the back-end, it enters a loop in which it waits until it receives a message from the back-end, then it performs the appropriate actions in response to that message. The dialogue is not strictly lockstep. Although the front-end usually sends a response message to the back-end, in some cases it does not. (For example, when the server sends a string that should be displayed to the user, the front-end displays the string, but does not send any acknowledgment back to the server.)

The network communication is handled by *block streams*, an implementation of Dave Clark's Unified Stream Protocol (USP). USP was designed to ensure a reliable, bi-directional stream of data across the network. Each message is transmitted in a *block*, which is structured to allow parsing into the message header and the body of the message, as described above. The block stream code on the Vaxes and MIT-XX is written in CLU on top of the TCP protocol. The block stream code on the IBM PCs is written in C.

2.3. Preprocessing for Scribe

The Scribe compiler is implemented to serve users on the same machine that it is running on. Consequently, it makes the following assumptions about its environment:

1. All files involved in the processing can be found on the machine that is running Scribe.

2. The user will read the messages displayed on the screen and respond to any questions posed by Scribe during processing.

These assumptions are not valid in the server environment of Dscribe. First, all files involved in the processing must be transferred from the front-end machine to the back-end machine (and they are renamed in the process). Second, it is the back-end process, and not the user, who initially receives Scribe's output messages and must decide how to handle them. Since the source code for Scribe was not available for us to alter, we had to find another way to reconcile these assumptions with the actual environment of the server. Our solution was to "preprocess" the manuscript file.

Preprocessing consists of two parts:

1. Making any files that are referenced (explicitly or implicitly) by the manuscript file available on the back-end, and

2. Modifying the manuscript file to reflect the new names of the files transferred to the back-end.

The preprocessor reads through the manuscript file in search of references to other files. As it encounters each reference, it arranges a file transfer, if necessary, and changes the reference to include the new name that the referenced file bears on the back-end. The referenced file may also have to be preprocessed in turn. (See Appendix III for details on how the preprocessor processes various Scribe commands.)

By doing this preprocessing, the server places all the files where Scribe expects to find them (in the server's working directory) and changes filenames in the manuscript file so that Scribe searches for files under their new names. If the server cannot find a file during preprocessing, it asks the user for another filename to replace it. The user then types in either a new filename or a carriage return to proceed. If the user does not supply substitute filenames for missing files, the server lists the missing files and halts without running Scribe. This precaution by the server ensures that *if* Scribe is called by the server to process the manuscript file, it will be able to find all the files it seeks and will not have to stop and ask the user for input. Since Scribe won't require any interaction with the user once it starts processing the manuscript file, the server can simply forward all screen output to the front-end as "display this text" messages without waiting for responses. We see then that preprocessing does reconcile Scribe's assumptions with Dscribe's server environment; assumption 1 is satisfied and assumption 2 becomes irrelevant.

3. Implementation

3.1. The front-end program

A front-end program must be written for every type of machine that wishes to access the Scribe server; therefore, the front-end program should be designed to be portable. Dscribe's front-end program is small and has a simple structure. When a user types "dscribe", the client program first opens a connection to the server program on the back-end machine. Then it enters a wait-action loop. The client waits to receive a message from the server. When a message arrives, the client dispatches to an appropriate subroutine which handles that type of message. The client also periodically checks to see if the user has typed a 'q' to quit. If this occurs, the front-end sends an ABORT message to the server and then waits for a message from the server to terminate.

In order to keep the client program small, we pushed as much computation as possible onto the back-end. The client program contains routines that send and receive messages, routines that deal with the user interface, and routines that must extract information from the front-end machine's operating system. All but one of the routines are application-independent. The routine that parses the JCL for Dscribe knows that the JCL may include Scribe option codes following the name of the file that is to be Scribed.

The client program is totally responsible for determining the user interface, including how messages are presented to the user and how the user should indicate his reply (e.g. via windows, menus, keyboard entries). This frees the server program from concerns about the user interface and what types of front-ends it might be dealing with. The client adapts the interface to best suit the capabilities of the host machine.

The client must also perform any tasks that require knowledge of the front-end operating system. For instance, if a character string needs to be parsed into a filename, the front-end program does the actual parsing and sends the result to the back-end. If the back-end had to do the parsing, it would have to know the filename syntax for every type of front-end machine it deals with.

3.2. The back-end program

In this section we give a detailed account of the Scribe server's actions in response to a Scribe request.

As mentioned in section 1.4, the Scribe server forks off a *handler* process to take care of each user requesting service. When the server creates a new handler, it passes to the handler its block

stream connection to the client. What used to be a connection between the "master server" and the client becomes a connection between the handler and the client. Network communication takes place over this connection.

Each handler makes several entries to a *Dscribelog* on back-end storage as it processes the user's request; this log is useful for monitoring use of the Scribe server and for tracking down problems during debugging.

The handler's first task is to determine which function the user wants. This is done either by examining the JCL of the Dscribe command line or by asking the user, as described in section 1.3. Assume here that the user has selected the Scribe function. We will consider three stages of the handler's processing: before Scribe is invoked on the manuscript file, while Scribe is running, and after Scribe has finished.

3.2.1. Before Scribe is invoked

The handler starts by making sure that an inactive copy of the newest version of the manuscript file is on the back-end. It does this by comparing the timestamp of the front-end's version of the file with that of the back-end's version. If the front-end's timestamp is later, a copy of the front-end file replaces the back-end's old version. If the back-end has a current copy, but it is active (being used in another Scribe run), another distinct copy of the front-end file is brought into back-end storage for the second Scribe to process. If the handler cannot find the file on the front-end, it requests a new filename from the user until either the named file can be found or the user types 'q' to quit.

Once the manuscript file has been transferred, the handler gets an up-to-date copy of the corresponding .aux file, if there is one, onto the back-end. The .aux file is an auxiliary file that Scribe saves information in from one run to the next. Because Scribe automatically looks in the working directory for the .aux file, the handler must get the .aux file onto the back-end, where it will be found by Scribe.

The next step is preprocessing the manuscript file, i.e. getting all referenced files into back-end storage and changing the manuscript file to include their back-end names. Section 2.3 describes this procedure, and Appendix III lists the various Scribe commands that require processing. The server keeps two copies of each file that is being used by an active Scribe run: one copy, called the "holding" copy, is the original version that was transferred from the front-end; the other copy, called the "prep" copy, has all the front-end filenames replaced by back-end filenames. Scribe is invoked on these prep copies.

The final step before actually invoking Scribe is translating the auxiliary file. All front-end filenames in the .aux file must be replaced with their corresponding back-end filenames. The server's name table provides the new names.

If, at any time before or during the Scribe run, some exceptional condition causes the handler to terminate, the handler first restores order by deleting all prep copies and deactivating all files related to the job.

3.2.2. While Scribe is running

The handler invokes Scribe by forking a Scribe process to work on the prep copy of the manuscript file. It then acts as an intermediary between the Scribe process and the client process on the front-end machine. The handler intercepts the output character strings that Scribe would normally display to the user. It translates any back-end filenames in these strings to front-end filenames (via the file table), and then forwards the converted strings to the front-end, where they are displayed. By parsing these output strings, the handler determines what the names of the Scribe output files are. If the Scribe process aborts for some reason, the handler checks to see if there is an error (.err) file that should be sent to the front-end machine.

3.2.3. After Scribe has finished

When the Scribe process has ended, the handler deletes all the prep copies of files related to the Scribe job from back-end storage, so once again, the server possesses only front-end versions of the files. The prep copies are deleted after each use because they cannot be reused. It is necessary to preprocess each file for each run in case the environment has changed. For example, referenced files may have different back-end names from one run to the next.

If the finished document produced by Scribe is a press or imp file, the handler asks the user if he wants to have it transferred to the front-end. (An exception occurs if the front-end machine is a PC; in that case, press and imp files are not transferred.) All other output files (.doc, .aux, .otl, .err, .lex) are automatically shipped over. Before the files are transferred, however, they must be translated; that is, all back-end filenames must be converted to front-end filenames. The handler consults its file table to perform this task.

If there is any trouble in transferring a file to the front-end, the file is cached in back-end storage. Otherwise, all of the Scribe output files, other than the finished document and the .aux file, are deleted from the back-end.

Finally, the handler deactivates all the files associated with this Scribe job. It then asks the user

if he wants the finished document to be printed out. (See the following section.)

3.3. Printing and pressmerging

In order to print and pressmerge files, the Scribe server makes use of already existing programs that perform these tasks. In the same way that it calls Scribe to format a file, the server calls the Pressmerge program to pressmerge a file, the Dover program to print a press file on the Dover printer, and the lpr program to print an imp file on the Imagen printer.

3.3.1. Printing

When the user selects the *Print* function of Dscribe, he supplies the name of a finished document produced by Scribe. The server determines what type of printer is required by examining the suffix of the filename. A filename's suffix must be ".press" or ".mpress" to be output on the Dover; the suffix must be ".imp" to be output on the Imagen.

The file that is to be printed must be stored on the front-end *and/or* the back-end machine. As long as the file is in back-end storage, it does not have to also be in front-end storage. This allows users to request printing of press or imp files that were generated by recent Dscribe runs but not sent back to the front-end. If, however, the file is not on the front-end, and the back-end copy is active (being used by another Dscribe run), the user is told to try later because there is no copy of the file that can be processed. This situation should occur very infrequently; however, if it does turn out to occur often, a method of queueing print requests could be added.

The Scribe server forks a Dover or lpr process to print the file. Flags are set so that the user's name and the front-end name of the file will be printed on the cover sheet of the printed document. The server intercepts the printing process's screen output, translates back-end filenames to front-end filenames that are meaningful to the user, and forwards the output to the front-end, where it is displayed.

3.3.2. Pressmerging

The Pressmerge program combines a main press file with one or more one-page press files. It is useful for inserting figures into a document. Suppose the user has a figure stored in the file "figure.press" and he wants to insert it into his main text file "main.mss". There are two ways to do this. The first way is to put an arrow at the place of insertion in the text of main.mss. The arrow looks like: <==<figure.press<. The second way is to insert a Scribe command: @Presspicture(height = 2 inches, file = "figure.press"). When the Scribe compiler processes this command, it allocates the specified amount of space for the figure and inserts an arrow "<==<figure.press<" into the press file.

When the user wants to pressmerge a file, he tells Dscribe the name of the main press file. Dscribe gets a copy of the main press file and preprocesses it. The preprocessor searches through the file, and when it encounters an arrow, it parses the name of the insert-file and fetches a copy of the insert-file to back-end storage. Because of the way the preprocessing is done, the Scribe server requires that the name of any insert-file not include any directories.

When preprocessing is done for a *Scribe* run, any referenced file is fetched to the back-end *and the reference is changed to include the file's back-end name*. The analogous thing does *not* happen in preprocessing for a Pressmerge run. The structured format of press files makes it difficult to substitute one string for another (the back-end filename for the front-end filename) in the main press file. Instead of altering the main press file, the server temporarily stores all insert files *under their front-end names* in back-end storage so the Pressmerge process can find them. When the pressmerging is finished, the insert files are "restored" to their back-end names in back-end storage. This scheme is feasible as long as the front-end filenames are legal filenames on the back-end machine. If a front-end program were to be written for a host machine whose filenames were not legal on the back-end machine, then a method would have to be devised to insert back-end filenames into press files so the Pressmerge process could find needed insert-files.

If any insert-files are found to be missing during preprocessing, the server presents a list of the missing files to the user and asks him whether he still wants to pressmerge the file. If the answer is yes, or if all insert-files are present, the server forks a Pressmerge process and forwards the screen output to the front-end machine. When Pressmerge is finished, the user has the option of having the new ".mpress" file transferred to the front-end and/or printed on the Dover.

4. Usage

Although client programs have been implemented for Vaxes, MIT-XX, and IBM PCs, we expect most usage of Dscribe to come from users on the PCs. Users on MIT-XX and the Vaxes already have Scribe available on their machines, and there is little advantage to using Dscribe over Scribe. The cost of the network I/O incurred by the server probably makes Dscribe run slower than local Scribe programs. However, implementing the XX and Vax front-end programs has had the beneficial side effect of getting the USP code written and debugged on these machines. This USP code will be needed for future services.

If usage of the server is heavy enough, the back-end program may be moved from MIT-OA to a larger, faster machine to improve the program's performance.

I. *Help* information for Dscribe

This is a copy of the information that is presented when a user selects the "Help" option from the Dscribe menu.

Dscribe, the Scribe server, can be used to Scribe a file or to pressmerge or print (hardcopy) a file. When you type "dscribe", a menu will be provided and you will be asked to select the desired function: Scribe, Print, Pressmerge, Help, or Quit. If you choose Help, this information will be displayed. If you choose Scribe, Print, or Pressmerge, you will be asked for a file name. In the case of Scribe, you may append some Scribe option flags to the file name. After you have entered the file name, Dscribe will do a little preprocessing and then invoke the appropriate program to operate on your file. The version of Scribe that is used is Scribe 4(1400)-1.

*** A note about Pressmerging *** -- The names of insert-files must not include any directories. For example, either of the following two lines in a Scribe .mss file would cause Dscribe to do unpredictable things:

```
<==</usr/foo/figure.press<
@Presspicture(height = 1 inch, file = "<foo>figure.press")
```

This implies that any insert-file must be in your working directory.

When the Scribe and Pressmerge functions have completed, you will be asked if you would like the press or imp file transferred back to your host machine. (If your host is a PC, you will not be given this choice; press and imp files will not be transferred to PCs.) All other output files (e.g. .doc, .aux, .otl, .err) will automatically be sent back to you. Finally, you will be given the option of having the output file printed. The printer (9th floor Dover or 5th floor Imagen) will be selected based on the filename's suffix, which must be .press, .mpress, or .imp.

If you would like to bypass the menu, do the following: (XX users should substitute a '/' for each '-' here):

- To Scribe a file "example.mss":
  dscribe example.mss -f -w
  (-f and -w are some of the OPTIONAL Scribe option flags.)

- To print a file "example.press":
  dscribe example.press -print
  (The appropriate printer will be determined from the file name's suffix.)

- To pressmerge a file "example.press":
  dscribe example.press -pressmerge
  (The output file will be named "example.mpress".)

To quit from Dscribe at any time, enter a 'q'. Dscribe may ask you to verify that you want to leave the program.

Dscribe can be run in background -- just redirect the output to some file.

--comments and complaints to BUG-DSCRIBE@OA

II. Message headers

Here is a list of the message headers used by the Scribe server in communicating with its clients. A brief explanation is provided for each header.

```
CHECKJCL          = 490   % Checking whether user's command line has a jcl
HASJCL            = 495   % Informing BE that user's command line has a jcl
DISPLAYSTRING     = 500   % Sending string to be displayed to user
DISPLAYCHUNK      = 505   % Sending string to be displayed to user without newline
QASTRING          = 510   % Sending question that requires string answer
REPLYSTRING       = 520   % Sending string reply
QAINTEGER         = 530   % Sending question that requires integer answer
REPLYINTEGER      = 540   % Sending integer reply
QABOOLEAN         = 550   % Sending question that requires boolean answer
REPLYBOOLEAN      = 560   % Sending boolean reply
REQUESTID         = 570   % Requesting host, user, and working directory names
SENDID            = 580   % Sending host, user, and wd names (as 3 strings)
REQUESTJCL        = 582   % Requesting jcl - dir, filename, suffix, # options, options
SENDJCL           = 585   % Sending jcl - 3 strings, integer, series of strings
REQUESTTIME       = 590   % Requesting FE timestamp for filename
SENDTIME          = 600   % Sending filename, boolean (found?), and timestamp
                          %    (as 6 integers)
REQUESTFILE       = 610   % Requesting transfer of file
SENDBYTEFILE      = 615   % Sending filename, bytecount, and bytes of file
SENDFILE          = 620   % Sending filename (as string) and file contents
REQUESTPARSE      = 622   % Requesting FE parse of filename string argument
SENDPARSE         = 625   % Sending parse of filename string
REQUESTCONCAT     = 626   % Requesting conversion of directory name to concatenatable
                          %    form
SENDCONCAT        = 627   % Sending converted directory name
REQUESTACK        = 630   % Requesting acknowledgment
ACK               = 640   % Acknowledgment
REQUESTACCESS     = 642   % Requesting access in given mode to given file
SENDACCESS        = 645   % Sending boolean indicating whether access is permitted
EXITING           = 650   % Notifying - about to exit
ABORT             = 660   % Notifying BE that user wants to abort
TIMEOUT           = 670   % Notifying FE that BE has timed out
ERROR             = 700   % Error
```

III. Preprocessed Scribe commands

This appendix lists the Scribe commands that are preprocessed, explains what they mean, and describes how the preprocessor handles each one.

- **@Value(keyword)** -- When Scribe encounters a @Value command with a keyword of Manuscript, FileDate, RootFileDate, or UserName, it inserts into the text the appropriate string value. These values change when files are transferred to the back-end. To handle this situation, the server inserts at the beginning of every manuscript file a series of @String commands which set the values of these keywords to reflect front-end filenames and other information from the front-end.

- **@Include(filespec)** -- This command instructs Scribe to insert the text of the file "filespec" at this point in the manuscript file. When the preprocessor encounters this command in the manuscript file, it makes a copy of "filespec" available on the back-end and adds "filespec" to the list of files to be preprocessed (since @Include commands can be placed in @Include'd files).

- **@Part(part-name, root "root-filename.mss")** -- If this command occurs at the beginning of an included file, it means that this file is to be processed as a part of a larger document, specified by "root-filename.mss". The server makes the root-file and its corresponding .aux file available on the back-end since they are both used in the formatting process. Later it preprocesses the root .mss file.

- **@Use(component = "filespec")** -- This command directs Scribe to look in "filespec" for the desired component. The server makes the appropriate files available on the back-end.

    - **@Use(AuxFile = "other.aux")** causes the file other.aux to be used as the auxiliary file.

    - **@Use(Bibliography = "other.any")** causes the file other.any to be used as the bibliography file.

    - **@Use(HyphenDic = "dict.hyp")** informs Scribe that the hyphenation dictionary is in the file dict.hyp. If the filespec has no extension, the default is .hyp.

    - **@Use(Database = "<Directory>")** -- This command directs Scribe to look in the specified directory, rather than in the user's working directory, if it cannot find a database file in the standard Scribe database. If a database file is referenced in the manuscript file (by commands @Device, @Make, @Style, and @LibraryFile), the preprocessor looks in the private database directory on the front-end and transfers the file, if it is there, to the server's working directory on the back-end. All transferred database files are preprocessed in turn because they contain @Marker and possibly other @LibraryFile commands.

- **@Device(device-name)** -- The Scribe compiler formats the finished document to be output on the device specified by device-name. The name of the corresponding database file consists of the first 6 letters of device-name, followed by the suffix ".dev".

- **@Make(document-name)** -- This command specifies the type of document that is being created. The name of the corresponding database file consists of the first 6 letters of document-name, followed by the suffix ".mak".

- **@Style(Keyword = value)** -- This command sets the style keyword Keyword to the specified value.

  - **@Style(References = RefType)** causes Scribe to determine the bibliography format from the contents of file RefType.ref in the database.

  - **@Style(FontFamily ffname)** specifies that the FontFamily ffname be used in printing the document. The corresponding database filename consists of the first 6 letters of ffname, followed by the suffix ".fon".

- **@LibraryFile(filespec)** -- If the user wants to include a file containing a set of definitions for his manuscript file (but no text), this command gives the file's name. The corresponding database file name consists of the first 6 letters of filespec, followed by the suffix ".lib".

- **@Marker(entry type, name, device qualification)** -- This command occurs at the beginning of each database entry in a database file. The preprocessor changes the name field to contain the back-end name of the database entry.

- **@Cite(codeword)** -- This command cites an entry in the bibliography file. If any @Cite commands exist in the manuscript file, the server must confirm the existence of the bibliography file on the back-end. If the .bib file is not specified in a @Use command, its default name is the same as that of the manuscript file with a suffix of ".bib".

- **@Comment(....)** -- Comments may contain Scribe commands which Scribe should ignore, so the preprocessor merely skips over the entire argument of this command without looking for commands within it.

IV. Where's the code?

This is a list of the files containing code for the Scribe server, organized by machine and directory.

MIT-OA: Directory /usr/chung

scribe – server.equ	*Equates file.*
scribe – server.lib	
scribe – server6.clu	*Main routines for server program.*
scribe – server6.bin	
scribe – server6*	*Executable server program.*
scribe – server6.xload	
handler47.clu	*Main routines for handler program.*
handler47.bin	
handler47*	*Executable handler program (forked by server program).*
handler47.xload	
utils47.clu	*Utility routines.*
utils47.bin	
cases42.clu	*Routines for preprocessing.*
cases42.bin	
printproc47.clu	*Routines to print and pressify.*
printproc47.bin	
– unhandled.clu	
– unhandled.bin	
filetable.clu	*Filetable cluster.*
filetable.bin	
nametable.clu	*Nametable cluster.*
nametable.bin	
queue.clu	*Queue cluster.*
queue.bin	
set.clu	*Set cluster.*
set.bin	
dscribe.spc	*Specs for front-end routines.*
dscribe8.clu	*Routines for Vax front-end program.*
dscribe8.bin	
dscribe8*	*Executable Vax front-end program.*
dscribe8.xload	

MIT-OA: Directory /fs/usr/local – clu (USP code)

/fs/usr/local – clu/blocks.clu
/fs/usr/local – clu/blocks.bin
/fs/usr/local – clu/btcp.clu
/fs/usr/local – clu/btcp.bin
/fs/usr/local – clu/host – name.clu
/fs/usr/local – clu/host – name.bin
/fs/usr/local – clu/port.clu
/fs/usr/local – clu/port.bin
/fs/usr/local – clu/chan – fudge.clu
/fs/usr/local – clu/chan – fudge.bin

MIT-OA: Directory /exe (executable)

dscribe*	*Same as /usr/chung/dscribe8.*
scribe – handler*	*Same as /usr/chung/handler47.*
scribe – serverd*	*Same as /usr/chung/scribe – server6.*

MIT-OA: Directory /etc

dscribe.hlp	*Information typed when user chooses Help option.*
dscribe – cleanup*	*C-shell script to clean up back-end storage.*
scribe.loc	*File holding command string which invokes Scribe.*

MIT-OA: Directory /adm

dscribelog	*Log for Scribe server use.*
dscribe/	*Directory where files are cached.*
dscribe/befiletable.beft	*File of info on files in BE storage.*
dscribe/benametable.bent	*File listing BEcodes corresponding to FE filenames.*

MIT-XX: Directory ps:<chung.server>

dscribe.clu	*Routines for XX front-end program.*
dscribe.tbin	
dscribe.exe	
dscribe.xload	
dscribe.equate	
dscribe.clulib	

MIT-XX: Directory ps:<mar> (USP code)

(Copies of these files also exist in ps:<chung.server>.)

blocks.clu	*Incomplete USP code.*
– create – file – form.clu	*.tbin file must be linked in.*
– chan.tasm	*.tbin file must be linked in.*

PC USP code and front-end code

Contact Bede McCall, Computational Resources, Laboratory for Computer Science.