Distributed Spooling in a Heterogeneous Environment

Bernhard Wagner Ciba-Geigy AG

ABSTRACT: Distributed spooling systems exist for many *homogeneous* systems. In this paper we describe the overall architecture of a distributed spooling system for a *heterogeneous* environment. This system allows exchanging jobs among UNIX, VMS, VM/CMS, and MVS operating systems connected by a network which provides the internet protocol suite (TCP/IP). The system's primary purpose is remote printing, but it is easily extensible to serve other purposes. Important design goals are portability, reliability, and user friendliness.

Some highlights of the implementation are presented as a case study. We focus on problems which arise especially in heterogeneous environments and explain why homogeneous solutions can hardly be applied to the heterogeneous case.

For each of the different systems, there is a more or less common implementation based on the client/server model [Svobodova 1985]. The user interface, however, is modeled specifically for each command language. A critical review of the project's issues and our experiences implementing a distributed heterogeneous system conclude the paper.

1. Introduction

The Scientific Computing Center of Ciba-Geigy AG is currently conducting a major project the main goals of which are:

- connection of a number of heterogeneous computer systems at the application layer [Zimmermann 1980]; and,
- offering services, for example, database access, workbench for chemical researchers, etc., to users of these computer systems.

One of the subprojects is the implementation of a spooling service which allows the execution of jobs, particularly print jobs, in any remote system of the network. It enables a user to keep track of her jobs and to get mail upon their eventual completion.

1.1 Motivation

The need for a distributed spooling system can be illustrated by the following scenario: A user working on system A (e.g., a VAX) receives a PostScript file via electronic mail which she wants to print. The only available PostScript printer, however, is attached to system B (e.g., a Sun). Sending the file to system B (e.g., via FTP or electronic mail) and printing it from there requires that the user has:

1. some working knowledge about B's command language and operating system; and,
2. a login account on system B.

In practice, both requirements have their drawbacks. Imagine, for example, the commands a VM enthusiast might enter when trying to delete a file from a UNIX system.

In order to reduce these problems we designed the distributed spooling system so that the user executes commands only in the style of the language she is used to. The *submit* command (see Section 2.2), for example, transmits a file to the desired system, prints it there, deletes the remote file afterwards, and sends mail to the submitter upon completion of the print job. The user needs no account on the remote system, she does not even need to know which operating system is used by the remote host.

The distributed spooling system is open in the sense that any command script may be executed remotely. (Such a script however, must be written in the style of the remote system.) To date, remote printing is implemented for many formats (e.g. PostScript, dvi, impress, pure ASCII) and plotters are connected to our system. We think that storing files on a special device (for backup purposes, e.g.) would also be a good candidate for implementation on remote facility. This openness also allows to add accounting and statistics facilities very easily.

When the distributed spooling subproject was begun in late 1987, we could find no commercially available product. Standardization efforts show the need for such a product: in February 1988, the working group JTC1 SC18/WG4 of ISO/IEC decided to start work on a *Document Printing Application* and to have this registered as an official *New Work Item*. The European Computer Manufacturers Association (ECMA) plans to publish a standard for a *Print Service* and a *Print Access Protocol* soon.

1.2 Heterogeneity versus Homogeneity

Remote printing facilities having similar functionality exist for homogeneous environments. All homogeneous solutions, however, are tailored to a specific computer system. As we will show, there are many good reasons not to extend an existing homogeneous implementation.

- One has to deal with different philosophies in operating systems. In one system for example, it is easy to spawn a new process which is impossible in another system.

- One has to deal with different philosophies in file systems, concerning for example the directory structures and access rights.
- The data representation is not necessarily the same on all machines.
- The number and types of privileges differ from operating system to operating system, which raises important security issues.
- In a heterogeneous environment, most involved machines are autonomous concerning their availability and accessibility. In particular, there is no central authentication instance.
- The code must be simple and understandable for people who are not specialists for every involved operating system, and it must not grow substantially if a new computer system is added. Otherwise, the distributed spooler cannot be maintained with reasonable costs.

1.3 Related Work

Possible approaches to distributed computing in heterogeneous networks are remote procedure call (RPC) (see e.g., Bershad et al. 1987; Wagner & Schaub 1987) and remote command execution via UNIX pipes (see e.g., Korb & Wills 1986). The RPC approach fails if a job's execution takes longer than the caller is willing to wait. UNIX pipes cannot be guaranteed in a heterogeneous environment.

Another approach would have been to extend an existing system. BSD-UNIX (LP) and VMS for example, offer remote printing, but only for homogeneous environments. *MDQS* [Kingston & Muuss 1982] and the *TCP/IP PrintServer* [Reid & Kent 1988; Kent 1988] are two other possible solutions to the problem of distributed printing.

Simplicity would require using already existing spool systems. The description of the existing approaches, however, makes clear that for us it was cleaner and less complicated to write a spooler tailored to our needs than undertaking the adoption of existing schemes.

1.3.1 LP Spooler

A line printer (LP) spooler which allows printing on remote machines is part of all derivatives of BSD-UNIX. On every machine there exists a daemon process for coordinating and controlling the spooling queues. If a request for spooling arrives, the daemon spawns a copy of itself to process the request, i.e. to print it or to transmit it to a remote daemon; the master daemon continues to listen for new requests. A file is submitted for spooling by using the *lpr* command. Under SunOS 4.0.3, the daemon and *lpr* both need superuser privileges.

The LP spooler is easily extensible to serve our needs in a UNIX environment, but implementing this approach would have implied incurring the time and effort to write a VMS symbiont and to spawn processes dynamically which is not an easy task in an IBM-system. Second, LP requires a specific protocol. Using LP would have forced us to implement servers understanding this protocol on every other system. Third, LP barely supports the retransmission of a job due to a communication error. These are the main reasons why we discarded the extension of the LP spooler from further consideration.

1.3.2 MDQS

The Multiple Device Queuing System (MDQS) was developed at the U.S. Army Ballistics Research Laboratory in the early eighties. MDQS is a general purpose queuing system for UNIX. In comparison with our system, it has some additional features like specification of a job's start time, prioritization, output limits, and modification of queue entries by the user. MDQS is very similar to LP.

Since MDQS runs on homogeneous machines only, it is not a solution for spooling in a heterogeneous environment. On the other hand, the central queue of MDQS is managed by a privileged daemon. As we did not want to use any privileges (see Section 3.1.4), we did not consider adapting MDQS to a heterogeneous environment.

MDQS could be easily integrated in our system. Any server which runs on a UNIX system may call MDQS when submitting a

job to a printer queue. In this way, our system could support multiple devices without adding extra code.

1.3.3 TCP/IP PrintServer

At the Western Research Laboratory of the Digital Equipment Corporation, a TCP/IP PrintServer was developed recently. It has many similarities and – concerning printing – basically the same functionality as our system. For example, both are running on top of TCP/IP and the administrator of our system corresponds exactly to the management client of the PrintServer. The main differences are:

1. The PrintServer does not provide any queuing. When a client wants to submit a job, it has to establish a TCP connection to the PrintServer and to keep it open until the job is finished. In our system, a client submits its jobs to the local system, so that unreliable communications and overloaded printers are handled in a user friendly way.
2. The PrintServer is designed to run on a free standing Ethernet connected printer. Although most of our printers are connected to an Ethernet, they are driven by specific hosts. For the sake of simplicity, our system uses the hosts' (commercially available) driver software, so we could implement our system on the hosts.
3. The PrintServer protocol requires the reliable transmission of bulk data – a problem for which we use the FTP protocol. Installing an FTP daemon on a printer, however, requires too much overhead in most cases.
4. Since the PrintServer is running on printer machines only, it is not extensible in the sense that it may remotely execute any arbitrary command script.

1.4 Environment

The base on which our project is implemented is a network with heterogeneous nodes connected by a homogeneous communication system at the transport layer. Its overall appearance is sketched in Figure 1. The network, at present, contains computers from four different vendors using five different operating systems:

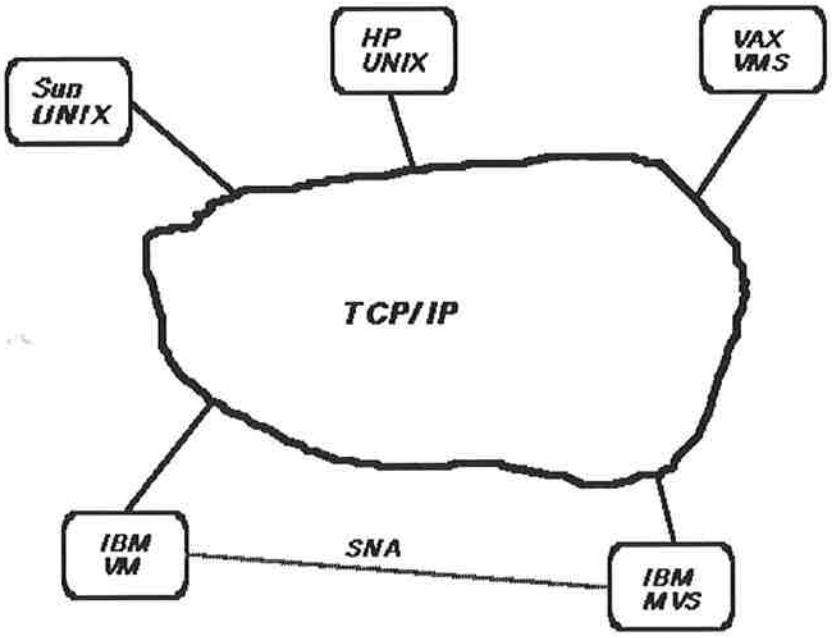

Figure 1: Environment

- Sun with UNIX (SunOS)[1]
- Hewlett Packard with UNIX (HP/UX)[2]
- VAX with VMS[3]
- IBM with VM/CMS[4]
- IBM with MVS[5]

1. The Sun/UNIX box represents four 3/280 servers and over forty diskless workstations (mostly models 3/50 and 3/60) connected by Ethernet. Most user home directories are mounted on all servers and workstations via NFS. We are now running SunOS 4.0.3, but the main implementation was done under SunOS 3.4.
2. The HP/UNIX box represents an Ethernet connecting one HP 9000/350 server and a number of HP 9000/340 workstations, all running HP/UX 6.2.
3. The VAX/VMS box represents a cluster of two VAXen (8700, 8820) plus one MicroVAX II connected via DECnet. The operating systems are VMS 4.7 and MicroVMS 4.6. The implementation of the VMS parts was done on the MicroVAX and the code transferred to the other VAXen later on. We are using Wollongong's WIN/TCP package, version 3.1.
4. The IBM/VM box represents one IBM 9370 running VM SP5. IBM provides a TCP/IP package for VM; we have actually installed version 1.2. Thus the IBM/VM is part of our TCP/IP network.
5. The IBM/MVS box represents a number of many kinds of IBM machines connected by SNA links and running MVS. There exists also an SNA link between the IBM/VM system and one of these machines.

1.5 Outline

The outline of the rest of this paper is as follows: in Section 2, the generic user interface of the distributed spooling system is described. This section is a part of the requirement specifications. In Section 3, some important details of the implementation are discussed and we present considerations which impacted the design. In Section 4, we discuss some subtle aspects of the design, such as security, and present some problems which showed up during implementation. The final section contains some lessons we have learned during the distributed spooling project.

2. User Interface

It is one of the most important aims of the distributed spooling system that a user can access it without leaving the philosophy of the operating system with which she is actually working. This makes the whole system extremely user friendly, as a user may execute jobs on a different system without learning anything about it. Thus we designed support for a generic user interface with a different "look and feel" for each command language.

2.1 Look and Feel

For example, printing the file *blabla* on the default printer of host *mist* and requesting notification (mail) upon completion is indicated by:

UNIX: `lprem -Hmist -m blabla`

VMS: `dprint /host=mist /not(ify) blabla.;1`

VM/CMS: `printr blabla text a (host mist mail`

MVS: `exec SPOOLCL 'file(blabla),host(mist),notify'`

Note that the command for MVS is a TSO-command which submits a batch job. The generic interface does not consider the differences in the syntax, but rather defines the functionality.

2.2 Generic Interface

When discussing the user interface, one has to distinguish between an ordinary *user* (henceforth simply called "the user") and the *administrator*. The latter plays a similar role for our system as the administrator for a database.

The administrator must have a login for each computer system on which the distributed spooling system is installed. The administrator owns all files related to the distributed spooling system and has read and write access to them. The administrator's identification and password are not fixed but may differ for each system.

A user has the functionality of three commands for communication with the distributed spooling system:

1. *submission*: used to submit a job to the system. With this command, the user may specify:
   - the (remote) host on which the job should be executed;
   - the (printer) queue in which the job is placed on the remote host;
   - the name of the local file which will be sent to the above mentioned host;
   - the file's format. At present, the system only supports the printing of character files, device independent files, and PostScript files. Below we will describe how the administrator can introduce more formats, i.e., more kinds of job handling;
   - several other parameters such as the job's name, title, and request for notification upon completion.
2. *listing*: used to show information about job.[6] If this command is used by the administrator, the jobs of all users are listed. By specifying a host, or a host together with a queue, the user gets the status of the distributed spooling server or of the queue at the remote site, respectively. This use of the

6. Only the jobs are shown which were submitted on the host on which the *listing* command is executed.

listing command fulfills a similar function to the *ping* command of the internet protocol suite. It allows one to determine if the distributed spooling system is installed at a specific host.

3. *killing*: used to remove a job[7] from the system before it is completed.

In order to introduce a new program for job handling, the administrator has to take the following steps:

1. Select a key word which corresponds to the syntax of the format parameter of the *submission* command.
2. At the server site, bind the keyword to a fully specified program or script by adding both to a file.

If a job using the new key word is submitted, the program bound to this key word is executed at the server site. Several parameters are delivered to this program, among them the name of the queue, the name of the job's owner, and a file with arbitrary content. Hence, it is possible that this file contains a command script which is eventually executed by the program on the owner's behalf.

One has to distinguish two classes of remotely executable tasks:

1. generic tasks which have a common design and a site specific implementation. An example for a generic task is printing.
2. specific tasks which are implemented on a specific site and are executed only there.[8] An example of a specific task could be doing some calculations on an array processor.

Our distributed spooling system may be extended to both classes of tasks.

7. The same restrictions apply for *killing* as for *listing*.
8. This class is commonly known as remote job entry.

3. Implementation

3.1 Boundary Conditions

Conducting a project involving several heterogeneous systems, one has to observe many restrictions. These restrictions are mainly imposed by security considerations, by resource limitations, and by the requirement for maintainability of the whole system. Our boundary conditions for the implementation of the distributed spooling system are:

3.1.1 Common Design

Despite the major differences of the several operating systems, we decided to make a common design for the following reasons:

- We have to show for only one design that it is correct and fulfills the requirement specifications. The correctness of our system was shown by walkthroughs with experts of every involved operating system.
- We were forced to concentrate on the most important aspects of a distributed spooling system and to omit all the fancy stuff which is easily implemented in one system, but can be realized only with excessive effort in other systems.
- A common design is not specific to one system. It should be understandable by people who need not be specialists for all the involved computer systems. This reduces the maintenance costs dramatically.
- The communication protocol has to be commonly designed anyway.
- A common design is a necessary prerequisite for a common implementation.

The drawbacks of this decision are that we could not use existing spooling systems. There is also a trade-off between maintainability and efficiency in terms of run time and memory requirements.

3.1.2 Common Implementation

Since we had a common design, we tried to write common code which can be compiled in all systems. We were quite successful: 10 modules out of 13 are used in all systems.[9] With the exception of the user interface, all modules were written so that they could be used in at least three systems.

The use of common code is on some systems not so efficient as it could be, but for a heterogeneous spool system we consider maintainability to be more important.

It is obvious that there is system specific code which requires that there be several implementations of the same module. The modules' interfaces however, are identical. This happens in 3 modules out of 13: the code using the TCP/IP interface of VM/CMS is completely different from all other systems.

There were no discussions as to which programming language to use – C is a company internal standard for this kind of project and compilers are available on all systems in our environment.

3.1.3 Portability

One of our most important aims was to design our system so that the same code (with only slight modifications) could be used on any other operating system. This goal of portability was reached for UNIX and VMS. In MVS, it appears to be attainable, as all of our test programs run there. Since we do not need to print on MVS, however, we implemented only the client part on that system.

In VM/CMS, we did not achieve the goal of portability. The description of our protocol (Section 3.3) shows that the protocol cannot be easily implemented if the server runs VM/CMS, because this operating system does not support sharable files. It is impossible to send a file via FTP to a user who has another process[10] running. Since we do not plan to use VM/CMS as a server, we have had no need to implement the server side there so far.

Further aspects of portability are discussed in Section 4.3.1.

9. Minor differences are handled within each module by compiler directives. This has the consequence of reducing the readability of the code.
10. In VM/CMS terminology: machine

3.1.4 *Privileges*

It is important that the distributed spooling system does not need superuser privileges for the following reasons:[11]

1. The administrator(s) would then have to have superuser privileges on all systems running the distributed spooling system. This is not in conformance with the company's security rules.
2. With superuser privilege, common caution, as well as mandated corporate procedure, requires a stringent safety review of the code and of all accessible programs, with a final approval by the managers responsible for each installation. With so many machines involved, and the general extensibility of the system, this becomes unmanageable.

We admit however, that the design and implementation of some aspects would have been easier if we had had superuser privileges. We will discuss these points in Section 4.2.1.

Having no superuser privileges also implies that we made no kernel modifications.

3.1.5 *Reliability*

Since a system such as the distributed spooler may be widespread over a big network, and must operate reliably without operator invention, robustness and simplicity are important. For the sake of maintainability, we also wanted to have a small system (which also increases robustness).

One aspect helping us to meet the goals of simplicity and maintainability is that all files containing information about the distributed spooling system are character files which may be modified using any simple editor. This saved us from developing supervision and maintenance tools.

11. In Section 3.2.2 we will show why we needed some, but not all, privileges in the VMS system.

3.2 Architecture

The architecture of the distributed spooling system is shown in Figure 2. It is based on the client/server model. A computer system can be a client or a server or both.

On the client side, the user has access to the distributed spooling system. Clients transmit jobs to the desired hosts and keep track of them. Servers accept jobs and requests about the jobs' state from any client and execute the accepted jobs. The main differences between clients and servers lie in the protocol (see Section 3.3).

Both sides each consist of two processes indicated by the circles in Figure 2. The user process circle represents all users who may concurrently access the spooling system. On each side, the processes have reading and writing access to a common spool file containing a description of every actual job known to the side.[12] The communications between clients and servers are implemented by protocols based on UDP [Postel 1980].

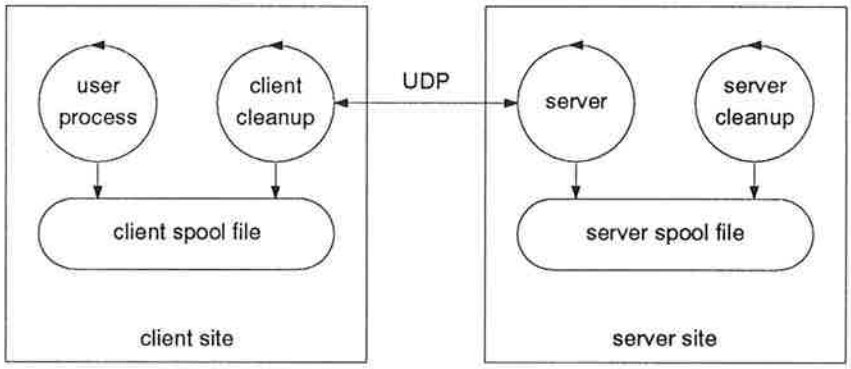

Figure 2: Architecture

3.2.1 Spool Files

A spool file consists of an administrative header and an entry each for all actual jobs. Because of the existence of the header, the file is never empty. All entries have the same length, which facilitates the internal organization of the file.

12. Obviously, if a computer system is both client and server, there exist two *spool files*.

An entry contains all details of a job. Besides the parameters given by the user, these details are the job's state, the date/time of submission, and a file name under which the user's file is stored. Figure 3 gives an schematic overview of the *spool file*.

Every job has a systemwide unique identification. It consists of the host name of the machine where the job was submitted and of a number which is locally unique to this host. In order to create such numbers, a client's *spool file* contains a value in its header which is incremented whenever a new job is written to this file.

3.2.2 Processes

The processes on each site are distinguished by their activation dependency: the user process and the server are event dependent, the cleanup processes are time dependent.

Whenever a user calls one of the commands described in Section 2.2, the local *spool file* is read and/or written. The client cleanup process checks the *spool file* periodically to see if there are jobs which require an action. New jobs, for example, are sent to the server process at the remote host. All network communication on the client side is done by the cleanup process. Because several users may attempt to access the *spool file* concurrently, we used a mutual exclusion mechanism. There is no communication delay

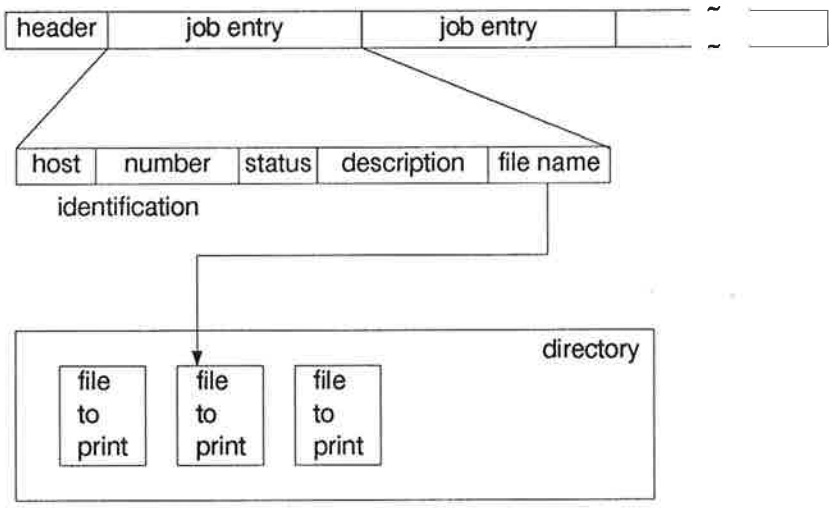

Figure 3: Spool file

during command execution, but trying to access the *spool file* may cause a short delay for the user.

The server process waits for packets from any client in an endless loop. When a new job is completely received (for the protocol, see Section 3.3), it is executed by the server and the outcome of the execution is written to the *spool file*. The server cleanup process checks the *spool file* periodically to find outdated jobs, which it removes. This helps to keep the protocol simple. The implementation of the server cleanup process did not require an extra effort, since the basic algorithm and the procedures for mutual exclusion were needed for the client side anyway.

The cleanup processes and the server are running as daemons [Lennert 1988; Saul et al. 1989] under the administrator's identification. In SunOS we could have used the *inetd* daemon instead which supports the installation of servers. Since there is nothing similar in other systems, we did not consider this approach.

A critical point is the access of user processes to the administrator owned *spool file*. In UNIX and/or VM/CMS, this is easily solved by *setuid* calls[13] and/or reader queues. In VMS, there is no solution other than giving the *bypass* privilege to the administrator (which contradicts the requirement of having no special privileges[14]).

3.3 Protocols

As stated in Section 1.3, the RPC protocol is not appropriate for distributed spooling. Because of the possibility of having to transmit large amounts of data, we introduced two logical communication channels: a data channel and a control channel.

For transmissions over the data channel, we chose the FTP protocol. The FTP program is invoked to transmit files directly from the client's spool area to the server's spool area. We chose FTP, because we wanted to use existing programs as much as possible, and because the simpler TFTP protocol is not available on all machines. Another reason for choosing FTP is that this

13. The programs are setuid to the administrator, not to root.
14. Similarly, a system wide lock of a file needs the *syslck* privilege in VMS.

protocol is appropriate for heterogeneous systems. It deals with all the file system and code differences of the two operating systems involved and it is reasonably reliable.

The use of FTP has a drawback: this protocol has no procedural interface. Calling a program from within a program as we did with FTP, and getting the result of the file transfer is a painful hack in most systems. In a homogeneous environment, we certainly had not used FTP, but rather a system specific program instead of (in UNIX e.g. *rcp*).

The easiest way to implement the protocols of the control channel would have been to use RPC. RPC, however, also was not available on all systems, so we had to implement our own protocols.

We designed simple handshake protocols, by which the client can send control requests to the server. The server executes an operation corresponding to each request and returns a response. All operations of the server are idempotent, so that we can build our protocols above UDP connections and, in case of errors, the client may retransmit a control request without problem.

We preferred UDP over TCP for mainly two reasons. First, because the cleanup processes deal with the uncertainties of the communication, we do not need the reliability which is inherently in TCP connections.

Second, the establishment of a logical connection (the *connect* call) is much more efficient in UDP than in TCP. The latter requires a packet exchange with the server. Since the distributed spool system may concurrently handle jobs for many hosts, the performance of connection establishment is an issue. The other solution would have been to establish the TCP connection for every job only once and to keep it active upon completion of the job.

The drawback of not using TCP – although it is available on every system – is that we have to deal with time-outs (see Section 4.2.2). Another solution would have been to introduce only one logical communication channel, and to use the TCP protocol for transmitting both data and control messages. In this case, however, we would have had to "reinvent the FTP wheel" without having reduced the cleanup processes substantially, since we have

to deal with other than communication errors anyway. Hence we discarded this approach.

We never switch the roles of client and server. That means that when a server has completed a job, it does not send the result to the client, but rather waits until the client asks about the status of the specific job.

If we had the client waiting for a packet containing the job's result at the server site, we could do that synchronously or asynchronously. The former blocks the client as long as the server executes a job whose duration is unpredictable. The latter requires the installation of a server process at every client site, which seems us to be too complicated. Piggybacking the job's result with packets of other jobs would be another solution for the asynchronous case, but it does not work for client hosts which rarely submit jobs.

An important aspect of the protocols is the structure of the packets which are transmitted. We have one `union` structure. It basically consists of a structured component for every possible request and/or reply. These components consist only of character arrays and of positive integers fitting into one byte, so that the conversion of one machine's representation to another machine's representation is easily done. We use 7-bit ASCII as the common representation for the transmission.

All user commands correspond to single control request with the exception of *submission*, which is a composition of the simple handshake protocol and FTP. An outline of the submission protocol is shown in Figure 4. It consists of several steps:

1. Prerequisite: the job has the status `waiting` on the client site. In normal case, this means that the job is not already in progress.
2. The client sends a *new job request* to the server using the simple handshake protocol. The client's request contains the job's identification and parameters. If the job is accepted, the server returns a file name, otherwise, a negative acknowledgment. At the server site, the job gets the status `waiting`.
3. The client sets the job's status to `in_transit` and transfers the job's file to the remote host via FTP.

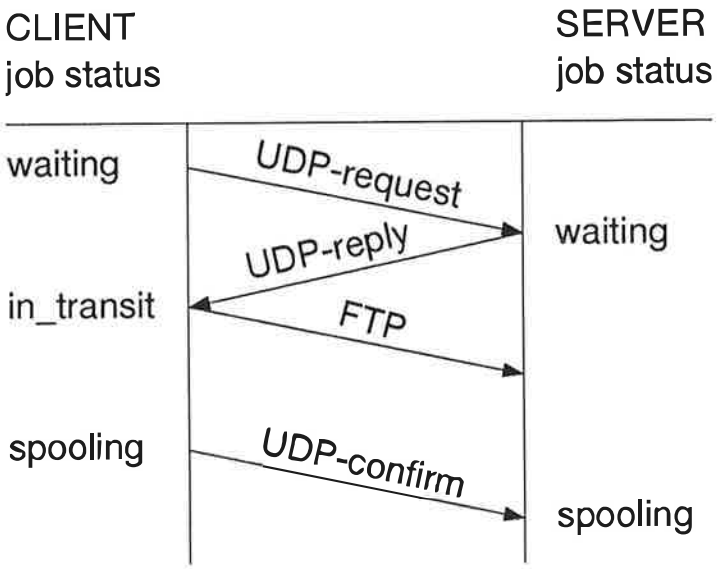

Figure 4: Submission Protocol

4. The client sends the result of the FTP transfer (*okay* or *not okay*) to the server in a single packet without waiting for a reply.
5. If the FTP transfer was okay, the client sets the job's status to `spooling`, otherwise to `waiting`. The server does the same when it receives the packet containing the FTP result.

The cleanup processes are crucial for the proper function of the protocol:

- On the client side, only the cleanup process executes the protocol. Thus, a user does not have to wait interactively for the termination of the protocol, which gave us more freedom in its design. If any step of the above protocol fails, the job on the client side stays in its current state. If a job stays in one state for too long, the cleanup takes appropriate action. For timing constraints, see Section 4.2.2.
- On the server side, the cleanup process removes all outdated jobs from the *spool file*. Hence the server does not have to keep track of its clients, allowing the use of the faster connectionless UDP protocol rather than TCP. The price we have to pay is the installation of two processes. Since the

function of the server depends on its cleanup process, a common parent starts both of them. If the server or cleanup dies, the parent restarts it, or kills the other one, depending on the reason for the first one's death.

- If the protocol is interrupted by an event caused from outside of our system (e.g. one of the participating processes dies), the cleanup processes detects this by timing out and it resets the job's status to `waiting` and/or removes the job. If a job is in `waiting` state, the cleanup process starts the submission protocol anew sooner or later.

In summary, the protocols and the cleanup processes assure that every site is self-contained. This means that no matter which errors occur during the protocol, a site can recover to a consistent state. During recovery however, a job may get lost.

3.4 Modules

The modules which build the distributed spooling system are shown in Figure 5. The left hand side of the figure including the middle row shows all modules which are used to build the client site, the right hand side including the middle row shows all modules for the server site. The modules in the four corners represent the main programs which describe the processes of the distributed spooling system (see Section 3.2.2).

The interface of each module consists of a number of procedures and type definitions which are known outside of each module. The arrows designate the import relations. The *base* module contains a bunch of useful procedures and is imported by all other modules; the corresponding arrows are omitted from Figure 5 for the sake of clarity.

Note that Figure 5 is asymmetric. At the client site, because all communication is conducted by the cleanup process, the *user* module does not import procedures from the *client protocol* module. The server site is structured by upcalls [Clark 1985]. The main program calls the server initialization procedure imported from the *server udp* module; this procedure has as parameter the protocol handling procedure imported from the *server protocol* module. To return an answer, a procedure of the

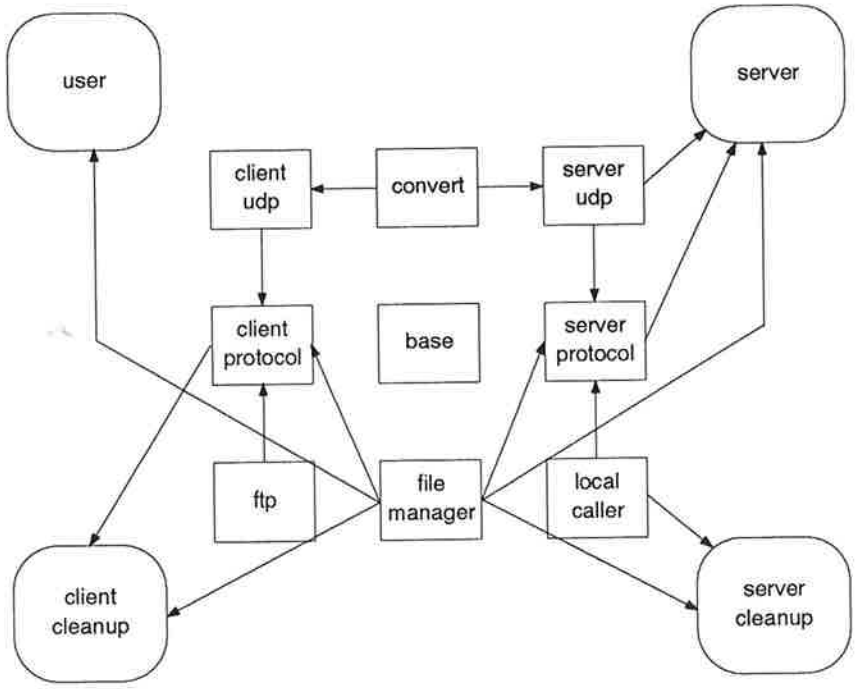

Figure 5: Modules

server udp module is called by the *server protocol* module. A more detailed explanation how to implement upcalls can be found in Wagner [1986].

Despite there being two spool files with different content[15] there is only one module for file handling. This comes about because the structure of both files is the same (see Figure 3). We implemented the file manager by using a set of procedure variables which are set at initialization time depending on whether the module is used by a client or by a server.

15. E.g. the structure of a client's entry is different from the structure of a server's entry.

4. Discussion

4.1 Security Aspects

In a system like the distributed spooler, which involves many heterogeneous computers and which is open to the world via any kind of network, data integrity and security aspects play an important role. First, we have taken care to use the least privilege possible. Second, the access to the system may be restricted, and the identification of all users is known.

4.1.1 Trusted Host and Network

We cannot burden the distributed spool system with all security issues. Hence we make trusted hosts and trusted network prerequisites.

The concept of trusted hosts means that users identify themselves by their local login password which is a sufficient authentication for the distributed spooling system. A more sophisticated user authentication, e.g. an authentication server, can be added to the distributed spooling if necessary.

Trusted network means that access to the network is possible only by trusted hosts. Therefore we do not need encryption on the network level and no one unauthorized may listen to the traffic.

Faking packets is theoretically possible, but it requires the knowledge of the exact structure of the packet which is nowhere described in public. The spool system checks the contents of every received packet. If a packet has wrong contents, its is discarded without further notice.

Every job description contains the host at which the job was submitted and the user who owns it. Both parameters are automatically set by the *submission* command. Hence, it is always possible to determine the real owner of a job. This knowledge may be used for accounting or for introducing access control on a user basis.

4.1.2 Access Control

The access to the distributed spooling system is controlled by the servers. Every server has a list of hosts (names and/or IP addresses) whence it accepts jobs. When the *new job request* packet is sent from a host which is not contained in this list, the server returns a negative acknowledgment.

Restricting the access by host rather than by user has as its motivation the avoidance of administrative overhead. Using the TCP/IP protocol suite, host names and/or IP addresses are known, but no user names.

For similar reasons, the control check is done on the server side, although this implies some unnecessary computations and network traffic. From the administration point of view, it is easier to restrict the access to a service at the point where the service is offered instead of where it is requested.

4.2 Problems

In implementing the distributed spooling system, we faced several problems which will be discussed in this section. Most problems are due to restrictions which were presented in Section 3.1.

4.2.1 Privileges

Due to the renunciation of superuser privileges, our system has some defects in the following points:

- Though the owner of a job is known at the server site, the job must run under the administrator's identification and with the administrator's privileges. This may cause some security flaws since the administrator usually has access to files which the user is not allowed to read and/or write. On the other hand, since it is undesirable that every (potential) user have an identification on a server, a more sophisticated solution is needed for this problem anyway. Using more privileges would give more freedom in finding such a solution.
- It is obvious that a user needs at least read permission for a file which she wants to print. During the *submission* command this permission is granted to the administrator (see

also Section 3.2.2). This implies that the administrator has to copy the file to one of her directories because, after the termination of the *submission* command, she can no longer read the user's file. The renunciation of superuser privileges prevents the faster and more elegant solution of storing a pointer to the file[16] at submission time and, later on, copying the file directly to the server via FTP.

- We cannot use a well-known port smaller than 1024 for the server. We found under SunOS that the port provided for the server may already be in use when the server is started, even if the port number is stored in the *services* file. The actual solution is to start the server at boot time and take a very large port number[17] which is most likely unused at that time. A better solution would be for the port numbers stored in the *services* file to be used only upon explicit request.

4.2.2 Timing

A well-known problem facing the designers of distributed programs is the choice for values of the time-out parameters when calling the procedure which implements the handshake protocol at the client site. In Birrell & Nelson [1984] it is proposed to use no time-outs at all, but this can not be applied to a wide area, heterogeneous network about whose reliability no assumptions can be made.

The issues which influence the choice of proper time-out values may be summarized as:

1. If the time-out value is too short, the client may decide that a job has not been transmitted even if no error occurred.
2. If the time-out value is too long, much time is wasted in the case of error.
3. There is no information available about the throughput of the (logical) line between any two given hosts. In an IP net, this throughput may vary by several orders of magnitude, depending on the paths involved.

16. In SunOS, a symbolic link.
17. Actually 15000.

For the sake of simplicity, we decided to deal with the time-out problem in the following way: For packet exchange, there is a fixed time-out value and a retransmission factor. Because of the dynamic environment of an IP net, both values are stored in a file so that they may be changed without recompilation. This allows easy experimentation to find the appropriate values.

Additionally, the time-out for the FTP transmission[18] is computed from the size of the file to transmit.

In order to make the system more reliable, the fixed time-out values are slightly longer than needed. Because of the architecture of our system (see Section 3.2), this only influences the client cleanup process, but not the user.

Another problem related to timing is the trade-off between response time and completeness of information in a distributed system. With the *listing* command, the user requires information which is generally stored on several machines. Since there is no maximum time limit for reliable data transmission, we decided to use only locally available information. Hence, the user will get a fast, but possibly incomplete response to the *listing* command.

4.2.3 Concurrence

Locking, which is stateful, contradicts the philosophy of NFS, which is stateless. We had problems with the lock daemon on SunOS as long as we tried to lock a file from several machines and then to modify the file. Finally, we decided to change the design by introducing empty files for locking purposes only.

4.3 Lessons Learned

The distributed spooling system is a utility program; it is very close to the operating system, since it contains real time aspects, concurrence, and uses the file system in a nontrivial way. Since this program is requested (among other things) to be portable to many machines and operating systems, and to be extensible for new user requests, its design and implementation are not obvious.

The lessons we learned from this project concern mainly compatibility and design rules. Let us examine both in greater detail.

18. In certain circumstances, FTP may hang indefinitely.

4.3.1 Compatibility

The programming language C is known to be portable. There are syntactic and semantic differences in system calls, however, which influenced the design of the distributed spooling system.

Let us take the system call *vfork* as an example. In UNIX, there exist the similar calls *fork* and (BSD) *vfork*, in VMS only *vfork*, and in VM/CMS nothing similar. The semantics of *vfork* in VMS differ from those in UNIX in the treatment of open files.[19]

There are even subtle differences between two types of UNIX: if the system call *gethostname* succeeds, it returns 0 in SunOS, but a non negative integer in HP/UX. And there is no help to detect semantic differences at compile time.

Another obstacle to compatibility lies in the file system philosophy, which may be different in every operating system. The richness of file formats in VMS, for example, cannot be mapped to UNIX or VM/CMS. The simple requirement of writing a character file and then reading its content requires a different open mode in every operating system. This brings up the question: "What is a default file which can be handled by a portable C program?"

Our solution for these and similar problems lies in applying the design rules described next.

4.3.2 Design Rules for Portable Programs

From our experience, the design of a portable utility program for a heterogeneous environment should contain the following steps:

- Top down path: Create a concept which contains all functions the system should, in the best case, have. Refine this concept until you have defined a number of module interfaces. Every interface should consist only of type definitions and procedure headers and be fully portable.
- Bottom up path: To implement the above defined interfaces, start with the low level procedures and try to implement them with common code. Make sure that all system and library calls which are used have the same semantics in the different operating systems. When the code is becoming

19. In VMS, if stdout is connected to a stream file, these standard streams will be redirected to the NUL device [VAX 1987].

too awkward because of special casing, go to the next higher abstraction level. Define the (partially) common functionality of the procedures, but implement them separately. Repeat this step for every abstraction level until all module interfaces are implemented.

- Redesign step: During the bottom up path, one may learn that for some machines some of the required modules can be implemented only at unreasonable expense. This implies a redesign for these machines on the base of the already existing modules.

5. *Conclusions*

We have described a user-friendly yet simple distributed spooling system for a heterogeneous environment. Its interface is adapted to each command language in which it is used. Although the system's main purpose is remote printing, it is open in the sense that it supports the remote execution of any job. Simplicity and portability were the main issues for the implementation. We showed that a large part of the system can be written in portable code, at least if one is willing to allow some restrictions.

It would be an interesting experience to install our distributed spooler on other, differing systems (e.g., on a VAX with Ultrix or on an IBM with AIX), but to date we have not had the opportunity of doing so.

Acknowledgments

Hans-Dieter Rhein implemented the user interface for VMS and helped me install the distributed spooler there. Wolfgang Reimann did the same for the VM/CMS and the MVS systems. Hans W. Barz was the project's supervisor. I would like to thank all of them for their valuable help.

Many thanks are owed to Bruce K. Haddon for carefully reading earlier drafts of this paper and for making many valuable comments which clarified the presentation.

References

Brian Bershad, Dennis Ching, Edward Lazowska, Jan Sanislo, and Michael Schwartz, A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems, *IEEE Transactions on Software Engineering*, SE-13(8):880-894, August 1987.

Michael Schroeder, Andrew Birrell, and Roger Needham, Experience with Grapevine: The Growth of a Distributed System, *ACM Transactions on Computer Systems*, 2(1):3-23, February 1984. Also in Xerox PARC technical report CSL-83-12.

David Clark, The Structuring of Systems using Upcalls, *Operating Systems Review*, 19(5):171-180, 1985.

Christopher Kent, TCP/IP PrintServer: Server Architecture and Implementation, Technical Note 7, DEC Western Research Laboratory, Palo Alto, CA, November 1988.

Douglas Kingston and Michael Muuss, The Multiple Device Queueing System, *Summer USENIX Conference*, Boston, 1982.

John Korb and Craig Wills, Command Execution in a Heterogeneous Environment, *SIGCOMM'86 Symposium on Communications Architectures and Protocols*, pages 68-74, Stowe, VT, August 1986.

Dave Lennert, How to Write UNIX Daemons, *UNIX World*, V(12):107-117, December 1988.

John Postel, User Datagram Protocol, RFC 768, August 1980.

Brian Reid and Christopher Kent, TCP/IP PrintServer: Print Server Protocol, Technical Note 4, DEC Western Research Laboratory, Palo Alto, CA, September 1988.

Mike Saul, Larry Lace, David Robinson, and Jerry Toporek, Development of a VAX/VMS Server, *SunTechnology*, 2(1):85-88, Winter 1989.

Liba Svobodova, Client/Server Model of Distributed Processing, In *Kommunikation in Verteilten Systemen I*, Eds. D. Heger, G. Krueger, Otto Spaniol, et al., pages 485-498, Berlin: Springer-Verlag, March 1985. Informatik-Fachberichte 95.

[VAX] *VAX C Run-Time Library Reference Manual*, Maynard, MA: Digital Equipment Corp., March 1987. Order Number AI-JP84A-TE.

Bernhard Wagner, Cipon: A Model for Distributed Systems, Ph.D. Thesis, ETH Zurich, 1986. Diss. ETH Nr. 7988.

Bernhard Wagner and Markus Schaub, Design and Implementation of an Extendible Distributed System, In *Kommunikation in Verteilten Systemen*, Eds. N. Gerner and Otto Spaniol, pages 216-228, Berlin: Springer-Verlag, February 1987. Informatik-Fachberichte 130.

Hubert Zimmermann, OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection, *IEEE Transactions on Communications*, COM28(4):425-432, April 1980.

[*submitted Nov. 29, 1989; revised March 29, 1990; accepted May 8, 1990*]