1986

Command Execution in a Heterogeneous Environment

John T. Korb
Purdue University, jtk@cs.purdue.edu

Craig E. Wills

Report Number:
86-593

Command Execution in a Heterogeneous Environment*

John T. Korb
Craig E. Wills

Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD-TR-593

April 14, 1986

Abstract

As a user's computing environment grows from a single time-shared host to a network of specialized and general-purpose machines, the capability for the user to access all of these resources in a consistent and transparent manner becomes desirable. Instead of viewing commands as binary files, we expect the user to view commands as services provided by servers in the network. The user interacts with a personal workstation that locates and executes services on his behalf.

Executing a single service provided by any server in the network is useful, but the user would also like to combine services from different machines to perform complex computations. To provide this facility we expand on the UNIX notion of pipes to a generalized pipeline mechanism containing services from a variety of servers. In this paper we explain the merits of a multi-machine pipeline for solving problems of accessing services in a heterogeneous environment. We also give a design and performance evaluation of a general mechanism for multi-machine pipes using the DARPA UDP and TCP protocols.

*This work was supported in part by grants from the National Science Foundation (MCS-8219178), SUN Microsystems Incorporated, and Digital Equipment Corporation.

1 Introduction

The Tilde project is concerned with computing problems in a distributed environment composed of heterogeneous machines [CKTM84]. The computing environment is heterogeneous with respect to processor type and operating system. As a user's computing environment grows from a single time-shared machine to a network of general purpose machines and dedicated processors, the need for the user to access all of these resources in a consistent and transparent manner becomes desirable. The user would like to have easy access to new resources as they become available without learning any new syntax or protocols to do so. Any changes to the "computing engine" should be reflected to the user in only two ways: (1) as an increase in the number of available services; or (2) faster response for old services that are now available on more machines. Our DASH (*D*istributed *A*ccess to *S*hared *H*osts) project is looking at how the user should perceive his Tilde computing environment and the tools he has to access it [Kor84].

In a time-shared computing environment, the user logs into a machine and interacts with a command interpreter, we use the term *executive*, to perform commands. In the UNIX operating system [RT74], each command is in fact a binary file that is loaded into memory and executed. In our model of a distributed environment the user interacts with an executive on a personal workstation connected to a local network, and views commands not as binary files, but as *services* provided by *servers* in the network. The user no longer is logged in at a particular host in the network, but accesses the computing

engine using the workstation to locate and execute services on his behalf. Breaking the command–binary file link has three principal advantages: (1) A user does not need to be concerned if a particular binary file runs on a particular machine. If a service is advertised by a particular server then that server knows how to provide the service. (2) Each server can decide what services it offers. This concept fits nicely with the idea of dedicated machines offering specialized services. (3) Service names are independent of file names and a different naming convention, taking into consideration the naming requirements for each resource, can be used.

In our *client-server* model, the user only needs to be concerned with *what* services he wants and not *how* they are provided by a particular server. If a service is offered by more than one server then the workstation can intelligently decide the "best" server to use. A simple heuristic, for example, is to choose the server with the lowest "load to computing power" ratio. Each server may provide its services as it wishes, perhaps by executing a binary file, or using a special attached processor to implement the service. The workstation may also provide services, and act as both a client and a server. In all cases the implementation details of a service are hidden from the user and he views his world as a set of services that his executive will locate and invoke on his behalf.

The user can access resources from a wide variety of machines in a uniform manner by viewing all commands as services. Executing a single service provided by any server in the network is useful, but the user would also like to combine services from different machines to perform complex computations. An example might be a numerical analyst

who generates some set of data on one machine, sends it to a specialized processor for solution, sends the results to a plot service on another machine, and finally gets the results as a plot on his own workstation. The user does not want to perform each of these operations separately, but instead would like to combine them into a single *pipeline* of actions.

Many systems allow remote execution of commands on other machines in the network [WPE*83,UNI83]. Others automatically schedule cpu intensive tasks, such as compiling or text processing, on lightly loaded hosts or idle workstations [Ber86,Hag86,TLC85]. But what these systems often lack is an efficient and simple mechanism for feeding the results of one command to the input of the next. In the worst case the user is forced to store intermediate results in a file and invoke each command in the sequence separately. This method is clearly not satisfactory. In the case of a pipeline mechanism on the user's local machine, the user can combine remote commands using local pipes. This solution is acceptable if the local machine has comparable computing power to the other machines executing the commands, but may become a bottleneck if the machine is a workstation in a pipeline of heavy data flow between powerful server machines.

We want a mechanism to provide pipes, but want them implemented in a general manner so pipelined services communicate data directly and these services can be used in a heterogeneous environment. This requirement leads to a multi-machine pipeline facility, similar to the pipe facility introduced by UNIX, but built on standard transport level protocols. Each service in the pipeline is a *filter* that reads from an (unnamed)

input stream and writes to an (unnamed) output stream. Extending this simple concept allows the user to compose computations of services from many machines as easily as he composes commands in the UNIX world. The use of multi-machine pipes also allows services from many machines to communicate data without the need for a common file system, which is an important concern in a heterogeneous environment.

In this paper, we look at a design for implementing an efficient mechanism for multi-machine execution of services, and compare an implementation of our mechanism with pipe facilities available in the Stanford V-System [CZ83,BBC*83], and the UNIX operating system.

2 Motivation

In [Zwa85], Zwaenepoel identifies two types of implementations for pipes in a client-server based system:

- Introduce a *pipe server* process between two clients that want to communicate over a pipe.
- Abandon the client-server paradigm for this mode of communication and have the operating system kernel implement pipes.

The advantage of the first mechanism is that it does not require any special kernel support thus making it easier to implement and modify. This solution also retains the client-server model, and if transparent interprocess communication facilities are available, then

no additional protocol is needed to support pipes. The disadvantage of this approach is the performance penalty that must be paid by not having a kernel implementation. The kernel approach has better performance, but abandons the client-server model, and requires a protocol for communication between kernels in the case of clients on different machines. In the paper, Zwaenepoel shows measured performance for the pipe server implementation to be 8-25% worse than the calculated value for kernel pipes in the V-System. He concludes that the implementation of a pipe server using message passing, the principal means of interprocess communication in V, is quite practical compared to the additional kernel and protocol complexity needed for a kernel implementation.

We are interested in these results because our initial prototype implementation for service location and execution ran on a SUN workstation using the V operating system. From our prototype we gained experience in using the V pipe server to provide pipes for combining services from different machines. Since the pipe server ran on the workstation, all data traffic between pipelined services was routed through the workstation, as illustrated in Figure 1. Each bi-directional connection is implemented with V message passing, with each outgoing line from the workstation carrying input data for the remote server, and each incoming line carrying both output and error data for the workstation.

When the amount of data was large, the pipe server became a bottleneck. Our design of a new protocol for handling multi-machine pipes was motivated by the desire to remove the workstation from the data flow path.

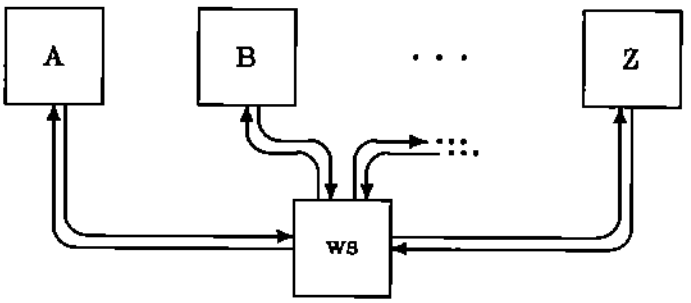

Figure 1: V Pipe Server Pipeline Execution

Our design is based on standard DARPA protocols and standardizes all interprocess communication at the transport level. Our mechanism is different from both described above in three respects:

- Like the V pipe server, our design follows the client-server model, but unlike the pipe server all data passed between services does not flow through the workstation. Instead, the workstation, acting as the client, communicates with each host in the pipeline to set up TCP [Pos81] protocol connections between services. When the services are executed, the data travels directly between service invocations using TCP connections, with the workstation supplying data to the first service, and receiving data from the last.

- The workstation sets up separate TCP connections with each server so that all error output from executing services is sent directly to the workstation. In the V system, all error messages and output are passed through the same pipeline.

- We do not use V interprocess communication for setting up the pipes, but instead use UDP [Pos80] packets.

The remaining portion of this paper looks at specifics of our mechanism including design, implementation, performance measurements, discussion, and conclusions.

3 Design

To remove the workstation from the data flow path, we have designed a general mechanism for creating a pipe directly between two executing services using DARPA protocols. We assume that each filter-style service in the pipeline follows the UNIX paradigm of reading from a standard input stream and writing to standard output and error streams. This mechanism requires the workstation command interpreter, the executive, to determine what server executes each service, and make connections to the first and last component of the pipeline to handle input and output, as well as a connection with each component to handle any errors that the service may produce. The pipeline configuration at execution time is shown in Figure 2. The thick lines indicate the flow of data through the pipeline, and the thinner lines indicate the path of errors from each component to the workstation.

The mechanism uses UDP datagrams for setting up the pipeline and TCP streams for the pipes between services. Each server machine in the network is required to execute a pipe server that listens at a well-known port for UDP requests. Once the executive determines what server will execute each service in the pipeline, it sets up the pipeline

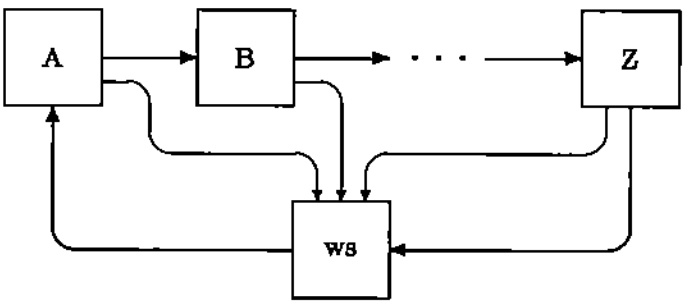

Figure 2: TCP Pipeline Execution

as follows:

1. Determine a local TCP port for the workstation to listen to for the output of the last service (server Z). Also determine a port to listen for errors from the last service.

2. For each service in the pipeline, beginning with the *last* one, and moving in reverse order, perform the following actions:

   (a) Send a UDP datagram to the well-known port on the server host providing the service. This datagram contains two TCP address (host, port)[1] pairs for the output and error streams. The datagram also contains the service name and command line arguments.

   (b) Wait for the server to return a datagram. Meanwhile, the server, listening on this well-known port, receives the incoming datagram, determines a port to

[1] A host's internet address and port number determine a unique endpoint for communication called a *socket*.

listen to for input and sends the port number back to the workstation. The server then must create a process to wait for a TCP connection on the input port. When this connection is made, the process must connect to the output and error ports given in the UDP setup datagram, and execute the service.

(c) Upon receiving a return datagram, use the port number given in this message as the output port for the preceding service in the pipeline and determine a new local port for receiving errors.

If a reply is not received from the server, then the executive times out and retransmits the UDP datagram. If a valid reply is not received from the server after a defined number of retransmissions, then the executive connects to the last successful (host, port) pair and closes the connections. This action causes all succeeding connections to close as well and "cleans up" the pipeline. An error is reported to the user.

3. After successfully setting up the first service in the pipeline, the executive starts a process that connects to the first server's (server A) listening port and directs all input from the workstation to this port. Then it starts a process to listen on the output port for data from the last command in the pipeline. The executive also starts a process to listen on each error port and collect any errors from the servers.

We based our design on the UDP and TCP protocols because we wanted a general mechanism that could be used by any clients and servers that understand these protocols

rather than limiting ourselves to just those that understand the V protocol. Our design requires each server to listen on a well-known port and be able to perform the services it advertises by connecting the service's input, output, and error to the given TCP connections. Otherwise, each server implements its services as it wishes.

This mechanism is very general and can be used by any clients and servers implementing the DARPA TCP and UDP protocols. We find this generality very appealing for work in a heterogeneous environment. We also feel that the design is an efficient mechanism for direct data flow between two services rather than using the client machine as an intermediary.

4 Implementation

The server portion of our design has been implemented on VAX, SUN UNIX, and RIDGE machines. The client portion has been implemented as a user program on our VAXes and incorporated into the command interpreter of the V operating system[2] running on SUN workstations. All machines are interconnected by a 10Mbps Ethernet.

V is a message-based operating system using a distributed kernel. A special purpose Inter-Kernel Protocol (IKP) is used for sending messages between processes on the same or different machines. Following the client-server model, the V environment has a server to manage the workstation display, an internet server (providing the IP and TCP protocols), a pipe server, and an executive server among others. Each of these servers usually

[2]We are currently using version 5.0 of V.

resides on the user's workstation. In addition, another server (the V UNIX server) executes on back end hosts to provide file access and remote command services.[3] Client programs can access any service by passing messages to the appropriate server.

Within this environment we have implemented a prototype that performs the following service location and execution functions:

- The service location mechanism caches a list of available services from all servers in the network. For VAXes, running the V UNIX server, the workstation communicates with the server using the V protocol, otherwise it uses a TCP-based protocol for communicating with servers on other machines.

- The executive periodically checks the load average on each of the server machines and stores this information locally.

- For each command, the executive chooses the "best" server for execution by checking its local cache for service availability and server load. To execute the service on the user's behalf, the executive communicates with the appropriate server using V or UDP protocols.

- For a pipeline of commands, the executive performs the same service location algorithm for each command, and uses either V or TCP pipes for passing the data between services. The type of pipe used is currently controlled by the user to facilitate taking measurements of pipe performance.

[3]The V UNIX server only executes on our VAX machines.

- The services provide by the V UNIX servers are executed in an environment that is maintained at the workstation and passed to these execution servers. The current implementation does not pass the environment to the TCP protocol servers. A default environment is used for services provided by these servers.

5 Performance Measurements

In this section we look at the performance of executing pipelined services using V pipes versus an implementation of the UDP/TCP mechanism we have described. The measurements were made by repeating each pipeline of commands and computing the average elapsed time. The time includes two parts: (1) the setup time to locate where the commands were to execute and create the necessary connections; and (2) the actual time spent waiting for the pipeline of commands to complete. For this experiment, the server to use for each service was specified to minimize differences in the comparison.

The setup for a V pipeline involves downloading a local (to the workstation) helper program for each remote service to handle input and output for that service. Also, for each pipe, the V pipe server is contacted to create a pipe and the appropriate I/O redirections are made. The executive waits until the last helper program in the pipeline exits.

Setting up a UDP/TCP pipeline requires determination of a local TCP port for the output of the last service in the pipeline, and a local TCP port for the error output of each service. For each component in the pipeline, beginning with the last, the executive

contacts the appropriate server and communicates as described. After creation of all connections, the executive downloads a local V program to connect to the input port of the first service, and to listen on all the local TCP ports. The executive waits until this program exits.

The results of setting up and executing services in each of these environments are given in Tables 1 and 2. The services chosen require little processing, so that time differences can be accounted for by the particular pipe implementation. The `cat` service reads the file given in the command line and writes the contents to standard output. If a file is not given then it simply copies its input to its output. The `wc` service counts the number of characters, words, and lines in its input and writes the result.

Each service pipeline was executed on files ranging in size from 10 bytes to 1 megabyte. All measurements were taken twenty times (ten or fewer times for the 1 megabyte pipelines) on lightly loaded VAX 11/780 machines. The bracket notation indicates the machine each service was performed on.

As a further comparison, we measured the performance of pipes on a UNIX system. The commands were executed on host A (a VAX 11/780) with remote execution of commands on host B performed using the `rsh` command. Commands were also executed on a SUN workstation running UNIX that used `rsh` to perform commands on hosts A and B. The `rsh` command logs the user into the remote machine and sets up the user's execution environment by reading from a standard startup file, then executes the

Pipeline Commands	setup time in secs (stddev) file size in bytes 10	1,000	100,000	1,000,000
[A]*cat* `file` | [A]*wc*				
V pipes	1.71	1.30	2.01	2.01
	(1.04)	(0.64)	(1.10)	(0.50)
UDP/TCP pipes	1.40	1.76	1.81	1.77
	(0.61)	(0.99)	(0.66)	(0.72)
[A]*cat* `file` | [B]*wc*				
V pipes	1.52	1.38	1.90	1.28
	(1.07)	(0.87)	(1.01)	(0.32)
UDP/TCP pipes	1.21	1.28	1.22	1.49
	(0.27)	(0.66)	(0.32)	(0.66)
[A]*cat* `file` | [B]*cat* | [A]*wc*				
V pipes	2.50	2.71	–[a]	–
	(0.90)	(1.11)		
UDP/TCP pipes	1.61	1.81	1.48	1.46
	(0.75)	(0.81)	(0.26)	(0.05)

[a]The V pipe mechanism did not complete.

Table 1: Pipeline Setup Times

Pipeline Commands	execution time in secs (stddev) file size in bytes 10	1,000	100,000	1,000,000
[A]*cat* `file` | [A]*wc*				
V pipes	3.06	3.24	23.71	196.69
	(0.74)	(0.56)	(1.08)	(2.36)
UDP/TCP pipes	2.72	3.34	7.86	48.58
	(0.60)	(0.24)	(0.38)	(1.04)
[A]*cat* `file` | [B]*wc*				
V pipes	2.99	3.74	18.28	168.75
	(0.87)	(1.05)	(0.91)	(16.80)
UDP/TCP pipes	2.32	2.42	6.26	31.39
	(0.41)	(0.50)	(1.82)	(2.05)
[A]*cat* `file` | [B]*cat* | [A]*wc*				
V pipes	4.46	4.03	–[a]	–
	(1.08)	(1.24)		
UDP/TCP pipes	3.16	3.35	8.75	60.21
	(0.42)	(0.47)	(0.79)	(4.45)

[a] The V pipe mechanism did not complete.

Table 2: Service Pipeline Execution Times

Pipeline Commands	total execution time in secs file size in bytes 10	1,000	100,000	1,000,000
[A]*cat* `file` | [A]*wc*				
VAX A	1.21	1.27	5.35	42.46
UDP/TCP pipes	4.12	5.10	9.67	50.35
[A]*cat* `file` | [B]*wc*				
VAX A	4.64	4.32	8.80	38.86
SUN UNIX	7.17	6.90	11.34	36.79
UDP/TCP pipes	3.53	3.70	7.48	32.88
[A]*cat* `file` | [B]*cat* | [A]*wc*				
VAX A	4.86	4.78	11.10	70.50
UDP/TCP pipes	4.77	5.16	10.23	61.67

Table 3: Pipeline Mechanism vs. UNIX

remote command. The remote command reads input, and writes output and errors to the originating host using a TCP protocol connection. To minimize the overhead of the `rsh` command, we removed the startup file for this experiment. The results are given in Table 3 as the average amount of time taken per pipeline.

One other performance consideration in comparing our mechanism with UNIX pipelines is the implementation of the UDP and TCP protocols in the V system. Just as the V system implements pipes with a pipe server executing outside of the kernel, the UDP and TCP protocols are implemented by servers executing outside of the kernel. Consequently, there is some performance penalty in not implementing other network protocols besides IKP in the kernel.

6 Discussion

Table 1 shows the amount of time to set up a two command pipeline is roughly the same for the two mechanisms, but for three or more commands the V pipeline takes longer to set up. This difference results from the executive downloading a V helper program to handle the input and output of each service in the pipeline. In contrast, the implementation of our design only needs to download one helper program regardless of the length of the pipeline, even though more UDP packets must be sent. Obviously, the cost of exchanging UDP packets with the servers is less expensive than downloading programs to the workstation.

The most obvious observation from the results is the performance penalty incurred in using the V pipe server on the workstation to pass data between services as shown in Table 2. For small amounts of data, the performance difference between our mechanism and V pipes is insignificant, but as the amount of data is increased the difference becomes large. For more data (>100,000 bytes) using just a single pipe, the execution time difference is between 3 and 4 times worse for V pipes than for TCP pipes.

The difference can be explained by looking at the path of the data in each implementation. For V pipes, each byte of data must travel from a server to the workstation and on to the next server in the pipeline. For our mechanism, each byte of data travels directly between the two servers. Not only does the V pipe implementation cause more data transmission, but it also introduces the workstation into the data flow path. When

the server machines are more powerful hosts, the workstation can become a bottleneck. These factors make our mechanism much better suited for data intensive pipelines.

Not surprisingly, Table 3 shows the performance of pipes on a single VAX machine is much better than for our mechanism because there is no overhead in setting up the pipe and all data is local to the machine. As the amount of data increases the relative difference between the two methods decreases. An interesting anomaly is the performance of our mechanism on a single pipeline of 1 megabyte of data between two VAXes. This pipeline outperforms even a single pipeline on a VAX. The difference might be explained by having two very lightly loaded VAXes instead of just one to handle this large amount of data. In fact, for our mechanism, a pipeline of two commands on different machines almost always outperforms a pipeline on a single machine.

The experiment to execute a pipeline between two VAXes from a SUN workstation running UNIX was made to test our conjecture that the pipeline would be slower because the workstation would be a bottleneck. For smaller amounts of data the pipeline through the workstation was slower, but for 1 megabyte of data, the workstation performs about the same as VAXes when we thought it would perform much worse. We concluded that the lightly loaded workstation is not a bottleneck because it still has the processing power to keep up with the incoming data. As the load on the workstation increases, or the pipeline becomes longer, we would expect the performance to deteriorate.

Using the rsh command to perform remote execution of UNIX commands causes the performance of low data pipelines to be slightly worse than our mechanism. This

difference is a result of the overhead to perform a login on the remote machine, even though the startup file had been removed for this experiment. In contrast our mechanism executed commands in a default environment. In previous experiments we used the rsh command without removing the user's startup file and measured 50% higher times for low data pipelines.

Future work on our mechanism will include passing information to each server to use in executing a service. For example, this information might include the client's user id or information about file access privileges. The server may use this information or use some default environment. This method is preferable to the rsh approach because a complete login does not need to be performed to execute a simple service, and some servers may provide services without requiring each user to have an account on the machine.

7 Conclusion

Coordinating the activities of many tasks in a heterogeneous environment is a difficult task for the user. The problem can be divided into two parts: (1) locating and invoking a command; and (2) combining many commands into a larger computation. As a solution to the first question we have modeled the system as a set of servers, each providing a set of services. The user then accesses the distributed environment through an intelligent agent, such as a workstation, that can interact with these servers to perform services on his behalf. The user does not need to be concerned with *where* services are located, or *how* they are accessed, but only needs to know *what* services he would like to use.

As a means for combining services in an easy and understandable way we have extended the UNIX pipe facility to coordinate the data movement between services on different machines. This approach provides the user with a simple, yet powerful, mechanism for composing services in a distributed world.

To implement multi-machine pipes in an efficient manner in a heterogeneous environment we have designed a mechanism based on standard DARPA transport protocols. Our design specifically addresses the issues of efficiency and use in a heterogeneous environment. It is efficient by removing the client machine from the data flow path. Existing facilities, such the V environment, allow remote execution but require the data to return to a server before proceeding to the next command in the pipeline. We demonstrated the weaknesses of this approach by showing performance penalties in using the V pipe server to handle large amounts of data.

Finally, our design is applicable to a heterogeneous system because it is not dependent upon a particular operating system or machine type. Even though much of the initial implementation has been done on UNIX, the design can be implemented on any machines that support standard DARPA protocols.

References

[BBC*83] E. J. Berglund, K. P. Brooks, D. R. Cheriton, D. R. Kaelbling, K. A. Lantz, T. P. Mann, R. J. Nagler, W. I. Nowicki, M. M. Theimer, and W. Zwaenpoel. *V-System Reference Manual.* Distributed Systems Group, Stanford University, Computer Systems Laboratory, 1983.

[Ber86] Brian Bershad. Load balancing with Maitre d'. *;login*, :32–43, January/February 1986.

[CKTM84] Douglas Comer, John T. Korb, Walter Tichy, and Thomas Murtagh. *The TILDE Project.* Technical Report CSD TR 500, Department of Computer Science, Purdue University, November 1984.

[CZ83] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[Hag86] Robert Hagmann. Process Server: sharing processing power in a workstation environment. May 1986. To be presented at the Sixth International Conference on Distributed Computing Systems.

[Kor84] John T. Korb. *An Overview of the DASH Intelligent Terminal Project.* Technical Report CSD TR 492, Purdue University, Department of Computer Science, September 1984.

[Pos80] J. Postel. User Datagram Protocol. August 1980. RFC 768.

[Pos81] J. Postel. Transmission Control Protocol– DARPA Internet program protocol specification. September 1981. RFC 793.

[RT74] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[UNI83] *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version.* Computer Science Division, University of California, Berkeley, August 1983.

[WPE*83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.

[Zwa85] Willy Zwaenepoel. Implementation and performance of pipes in the V-system. *IEEE Transactions on Computers*, C-34(12):1174–1178, December 1985.