The Deputy Mechanism for Transparent Process Migration

Yuval Yarom

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Supervised by Prof. Amnon Barak

Acknowledgment

I am greatly indebted to Prof. Amnon Barak for his help and guidance during all the stages of this work. His advice and suggestions greatly influenced the shape of this thesis.

I would also like to thank Amnon Shiloh for sharing with me his immense experience in developing distributed operating systems, and to all the Distributed Operating Systems lab. in the Hebrew University.

Special thanks go to Big Apple Pizza for their substantial support in times of need.

This research was supported in part by a grant from the Ministry of Defense and in part by an equipment grant from Intel Semiconductors (Israel).

Contents

Chapter 1

Introduction

The most important goal of distributed operating systems is the ability to transparently share the resources of several machines. In order to achieve this goal the system must provide a mechanism for using the various processors of the system. Process migration is one of the common mechanisms for CPU sharing.

The Process Migration mechanism enables the system to move a process from one Processing Environment (PE), and continue its execution in another. However, process Migration alone does not provide transparency: a process must interact with its environment, and for transparent sharing of the system resources, process migration must be backed by other mechanisms.

This work presents the deputy mechanism for supporting transparent CPU sharing through process migration. This mechanism is based on the observation that the main body of the process is site independent, and can be transparently executed in any PE, while only a small part of it is site dependent, and must be executed in a specific PE. To achieve transparent process migration the process is divided into two parts: the deputy, which contains the site dependent parts of the process, and the body, which contains the rest. These two parts maintain a communication channel between them. The body can be migrated, and executes normally on any PE, until it attempts to execute a site dependent operation. Such an attempt is intercepted by the kernel at the PE the body is executing on, and forwarded to the deputy. The deputy then executes the operation on behalf of the body, and returns it the result.

Related Work

Several systems support transparent process migration. Mechanisms used to achieve transparency vary between the various systems.

The Condor package [BLL92,Lis92] is a package that supports both checkpointing of processes and process migration. The most notable feature of Condor is the fact that it is implemented outside the system's kernel. As such Condor is highly portable, but it suffers both in performance and in transparency. The major limitations are for multiprogramming; no creation of new processes is allowed and interprocess communication is not supported. Whatever transparency supported is achieved by an assigning for each process a *shadow* process. The shadow is responsible for executing the process's system calls. The shadow differs from the deputy by being used only for the execution of system calls, while the deputy also acts as an alternate of the process, and forwards asynchronous events to the body.

Sprite [OCD88,Dou89] is a distributed operating system developed in the UC Berkeley. Sprite supports a distributed file system, thus making all files accessible at each PE. As communication mechanisms are implemented on top of the file system, interprocess communication is also globally available. The existence of this file system makes transparent process migration much simpler. Only a small number of system calls are site dependent, and these are forwarded to the home node of the process to be executed there. Sprite, unlike Condor, does not have a shadow for each process. Instead a single server at each PE serves all site dependent system calls.

The MOSIX operating system [BGW93] is a distributed operating system that supports transparent process migration and dynamic load balancing. The MOSIX kernel is divided into three parts: the lower kernel, which is site-dependent, the upper-kernel which provides the environment for user processes and is site-independent, and the linker which is used for the communication between the other two parts. This structure makes all system entities globally available to each PE. When a process attempts accessing a resource the upper-kernel at the PE that hosts the process forward the request via the linker to the lower kernel at the site the resource is in. As all resources are globally available, all system calls are site-independent, and process migration is transparent.

Amoeba [MRT90] achieves location transparency by requiring that all interaction between the process and its environment is done through a remote procedure call (RPC) mechanism, which provides the Amoeba equivalent for a system-call. The kernel at each PE is responsible for locating the accessed object, and for forwarding the RPC to it. As such, all resources are globally available

to processes, and RPCs are site-independent.

Organization of this Thesis

The second chapter presents the deputy concept. The third chapter discusses the communication protocol used between the deputy and the process body. The implementation of a prototype of a deputy-based system is described in Chapter 4. Chapter 5 presents some performance results.

Chapter 2

The Deputy Concept

This chapter presents the main concept developed in this thesis—the *deputy*. The deputy is that part of a process that cannot be migrated, and must reside in the home of the process, i.e. the PE in which the process was created. This splitting of a process is used for achieving transparent process migration.

The first section discusses the UNIX concept of a process. Transparent process migration is discussed in the next section, and the idea of splitting the process to achieve transparent process migration is presented in the last section.

2.1 The UNIX Process

A *process* is the execution of a program. In the UNIX operating system [Bac86,LMKQ89] the process is the basic computational unit. The system creates the illusion of concurrent execution of several processes by scheduling the CPU and memory between the processes.

A UNIX process has two contexts, the user context and the system context. The user context of the process contains the program code, stack and variables. The system context contains description of the resources the process is attached to, and a stack for the execution of system code on behalf of the process.

The process interacts with its environment using two mechanisms—system calls, and signals. The following subsections describe these mechanisms.

2.1.1 system calls

System calls are the mechanism which enables a UNIX process to operate on a resource. This mechanism relieves the programmer from the need to program a file system or a device driver. In this sense a system call is similar to a library function. In addition, by preventing direct access to resources, the system can preserve the integrity of such resources.

Executing of a system call consists of the following stages:

- System call arguments are stored in the runtime stack. These arguments include the user supplied arguments, and a number identifying the system call.
- The process transfer control to the system. After this stage the process is said to be executing in system mode.
- The system copies the arguments from the user stack. Arguments are copied to prevent their modification during the system call execution.
- The function that executes the system call is invoked. The address of the function is taken from a dispatch table using the system call identifier as an index.
- When the function returns the system arranges for the receipt of its result by the user's code.
- Finally, the system returns control to the program, and the process returns to user mode.

During the execution of a system call, the system might have to transfer data from the user context to the system context. For example, when the process reads from a file the arguments it gives to the system include and identifier of the file to be read, a pointer to a buffer in the user's address space, and the size of this buffer. The system then reads the data from the file and has to copy it to the user-supplied buffer. As transferring the data between address spaces is a machine dependent operation, all such data transfer is done through an interface provided by a small number of functions, e.g. *copyin()* and *copyout()*.

2.1.2 Signals

In order to inform the process of the occurrence of exceptional or asynchronous events, the system employs the signals mechanism. Whenever such an event occurs the appropriate signal is sent to the process. Exceptional events informed include access to an illegal address, attempt to execute

an illegal instruction, and similar events. Asynchronous events are events like the arrival of data on a communication port, or the user's hitting the 'ctrl-C' key to terminate the process.

Signals are modeled after hardware interrupts. Just as hardware interrupt has an interrupt handler associated with it, so does each signal have an action to be taken when the signal is delivered. The action associated with a signal ranges from simply ignoring the signal, to executing a special function in the user's program, to the termination of a process. In addition the process may elect to block arriving signals and to delay the execution of the action associated with them.

Sending the signal to the process consists of two stages. The first stage is posting the signal. This stage is executed in the context of whatever process the system was executing when the event occurred. First the system checks what the action associated with the signal is. If it is to ignore the signal, the signal is simply discarded, and nothing has to be done. If the process does not ignore the signal, the system notifies it of the signal's occurrence. Finally the system wakes the process up if it is sleeping and needs to be wakened.

The second stage is the signal delivery. In this stage the action associated with the signal is taken. Just before the system returns the process to user mode, either after serving a system call, or after handling a hardware interrupt, the system checks if any signals the process does not block were posted to it. If it finds such a signal it arranges for the action to be taken.

2.2 Process Migration

A distributed system is able to share the CPU resources of the system by using a facility known as *process migration*. To achieve this sharing, the system takes a process from one Processing Environment (PE), and continues its execution on another.

One of the major features required in process migration is transparency. The functional aspects of the system's behavior should not be altered as a result of migrating a process. Achieving transparency requires solving two problems, namely: the *rendezvous problem*, and the *location transparency*. These problems are described in the next subsections.

2.2.1 The Rendezvous Problem

The rendezvous problem consists of locating a migrating entity. Suppose, for example, that a process creates a child process, and asks the system to inform it of the termination of the child. By the time the child process terminates both processes might have migrated several times. The

system at the PE on which the child process terminates must be able to locate the original process to inform it.

One solution to this problem is assigning to each process a home PE. Whenever the process migrates it informs its home PE about its new location. When any entity on any PE wants to access the process it contacts the home, gets the location of the process, and can now access the process. To reduce the overhead incurred in contacting the home, the location of a process can be cached in the accessing PE, and the home has to be consulted only if the cache is empty or is invalid.

The decision on which PE is the home of a process differs between systems. In the MOSIX operating system the PE on which the process is created is the home of the new process. On Sprite the home of the new process is inherited from the creator. Both systems encode the home PE number in the process ID. When access to the process is required, its home identity can then be extracted from its ID.

2.2.2 Location Transparency

Location transparency consists of hiding the fact that the process has migrated from the process itself. As long as the process does not interact with its environment it can not become aware of the migration. When the process interacts with its environment (via a system call) the system must ensure that the result of the interaction does not depend on the site the process currently executes in.

Location transparency can be achieved by two methods. The first approach the system makes all resources accessible to each PE. The system at the PE is responsible for contacting the site the resource is physically on. Employing this method creates an identical environment in each PE, and thus the result of system calls is independent of the site they are executed in. MOSIX [BGW93] is a good example for such a system. The linker in MOSIX hides the environment details from the upper kernel. When the system call executes in the upper kernel, it is not aware of the location of resources. The linker transparently redirects accesses to the resource to the correct site, thus making all system calls site independent.

The second approach for achieving location transparency is forwarding the system call to the home PE of the process, where it will be executed. For this approach to work, the home must be inherited, otherwise a created process environment might differ from that of the creating process,

and the migration will not be transparent. The Condor package [BLL92,LiS92] uses this method. For each process Condor creates a *shadow* process in the home machine, which is responsible for executing the system calls. Condor is implemented outside the system's kernel. As such, it cannot implement all system calls, and is not completely transparent.

Sprite [OCD88] uses a hybrid approach. Most system calls are site independent due to the distributed file system it provides. Those system call which are site dependent are forwarded to the home PE of the process to be executed there by a special server that serves all environment dependent system calls on the home PE.

2.3 Splitting the Process

The concept of the deputy of a process is based on the observation that only the system context of a process is site dependent. Generally the idea is that the user context of a process is migrateable, and, since its interface to the system context is well defined, it is possible to intercept every interaction between the user and system contexts, and forward this interaction across the network.

In order to migrate a process we divide it to two parts: the *body* which has all of the user context and almost none of the system context of the process, and the *deputy* which has the rest of the system context, and no user context (see Figure 2.1). The user context and the part of the system context that is associated with the body are site independent. The body can, therefore, be migrated to any PE in the system. The deputy has the site dependent part of the system context of the process, hence it must remain on the home PE of the process. The two parts are connected by a communication channel, on which interaction between the two parts takes place. Figure 2.1 shows an unsplit process (process A), and a split one (process B).

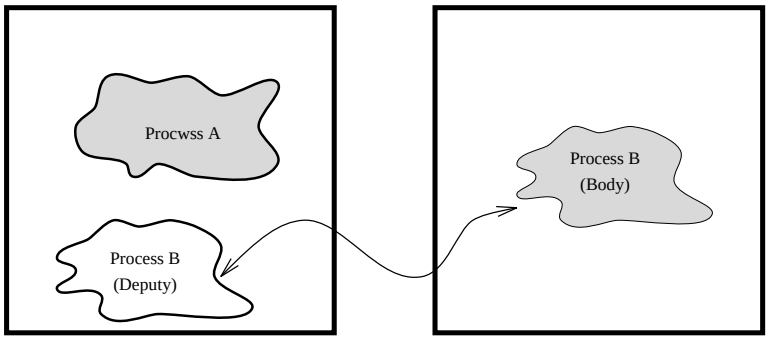

Figure 2.1: *Unsplit, and split processes*

Location transparency is achieved by forwarding site dependent system call to the home PE. System calls executed by the body are intercepted by the remote PE's kernel. If the system call is site independent it is executed in the remote PE, otherwise, the system call is forwarded to the deputy. The deputy executes the system call on the home PE, and sends the result back to the body, which then continues executing the user's code.

The deputy mechanism is a variant of the home PE solution to the rendezvous problem. Instead of providing the location of the process, the deputy provide a channel to interact with the body. The kernel at the home PE inform the deputy of asynchronous events, the deputy checks if there is any action to be taken, and inform the body if so. The body checks the communication channel for reports of asynchronous events in the same locations a standard UNIX process checks for signals.

We find the deputy concept a very appealing method of achieving transparency. The reasons for this are that it provides a simple and clean solution to the transparency problem. This solution is robust, and will not be affected even by major modifications to the system. Some modifications, such as supporting a distributed file system, might even improve the solution by making many system calls site independent. Finally, the solution seems to be portable. It relies on almost no machine dependent features of the system, and thus will not hinder system porting to different architectures. In fact, it is not relying much on UNIX specific properties, and seem to be exportable to other time sharing operating systems.

This solution has, however, some overhead on the execution of system calls. For example, as Figure 2.2 shows, data sent from one process to another has a two hops over the network, instead of just one hop. Additional overhead is incurred due to the need to hold enough state at the home PE, and to keep a dedicated communication channel. Furthermore, as the number of processes increases, the home PE might become the bottleneck of the system, thus putting a limit to scaleability.

We believe that all of the problems raised above can be solved by additional mechanisms. These mechanisms will make some of the resources globally available, thus reducing the number of site dependent system calls and improving the overall performance. Given the current memory prices, the memory requirements overhead due to the need to keep the communication links, and the deputies can be neglected. Idle deputies do not require much CPU resources, and with reducing the number of site dependent system calls, the deputies will be idle most of the time. The computational overhead in handling many communication channels can be solved by methods similar to those described in [MKD92].

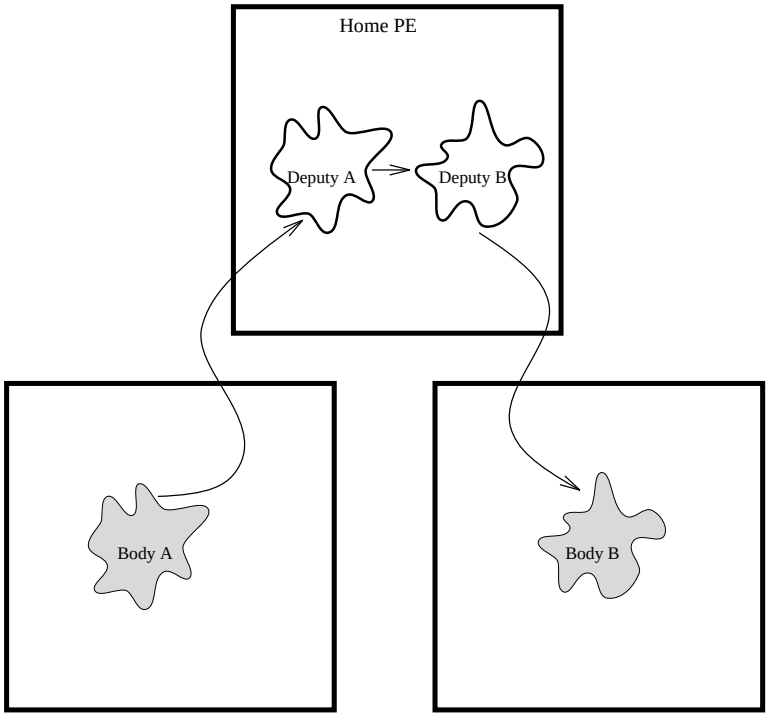

Figure 2.2: *The path of data sent from process A to process B*

Chapter 3

Communication

One of the major concerns in a distributed system is the protocol used by communicating entities. This chapter presents some of the standard communication protocols we had access to, and our choice of protocol for the intraprocess communication.

The system we worked on supports several communication protocol which belong to different families. These protocols include the User Datagram Protocol (UDP), Transmission Control Protocol (TCP), and Internet Protocol (IP) of the Internet Protocol suite; the Sequenced Packet Protocol (SPP), and Internetwork Datagram Protocol (IDP) of the Xerox Network System architecture; and the Transmission Protocol levels 0 and 4 (TP-0, and TP-4), Connectionless Transmission Protocol (CLTP), and Connectionless Network Protocol (CLNP) of the ISO's Open Systems Interconnect (OSI) communication standard.

As the OSI protocols implementation is still evolving, and the Xerox NS protocols definition is presently unavailable, the choice of communication protocol for implementing the link between the deputy and the process naturally fell on the Internet Protocol suite.

Section 1 presents the Internet Protocol suite, and Section 2 describes our choice of protocols for the intraprocess communication.

3.1 The Internet Protocol Suite

The Internet Protocol suite is a set of communication protocols initially developed by the Defense Advanced Research Projects Agency (DARPA). Like most networking protocol suites, the Internet Protocol suite is a layered set of protocols. Generally, there are 4 layers of protocols:

- application protocols such as File Transfer or Remote Login
- transport protocols such as TCP and UDP
- the Internet Protocol
- network interface protocols for hardware networks such as Ethernet or FDDI.

These layers roughly correspond to the application, transport, network, and data link layers of the OSI reference model, respectively.

Application layer protocols are too specialized to be used as the basis for the intraprocess communication. This communication requires some facilities that are not supported by IP, e.g. data demultiplexing, and reliability. As such, IP (and the network interface protocols) can not be used for our purposes. The choice is, therefore, limited to the standard transport protocols, TCP and UDP, which are described below. The following protocols are also described: Reliable Data Protocol (RDP), Remote Procedure Call (RPC), and IP itself.

3.1.1 IP

The Internet Protocol [Pos81a] is one of the major protocols in the Internet Protocol suite. It corresponds to the network layer in the OSI reference model.

The transport protocols pass to IP datagrams, and IP is responsible for transferring the datagrams to the destination host. At the destination host the datagrams are passed to the transport protocol they belong to.

IP is based on the "catenet model" [Cer78]. A catenet is a collection of independent networks interconnected by gateways. Figure 3.1 shows an example of a catenet comprised of two networks interconnected by a gateway.

IP hides the network structure from the upper layer protocols. It provides the following communication services:

- *Fragmentation and reassembly.* Underlying network hardware may be incapable of transferring large datagrams. IP fragments datagrams that are too large for the network hardware, sends all fragments as separate datagrams to the destination host, and reassembles the fragments at the destination host.

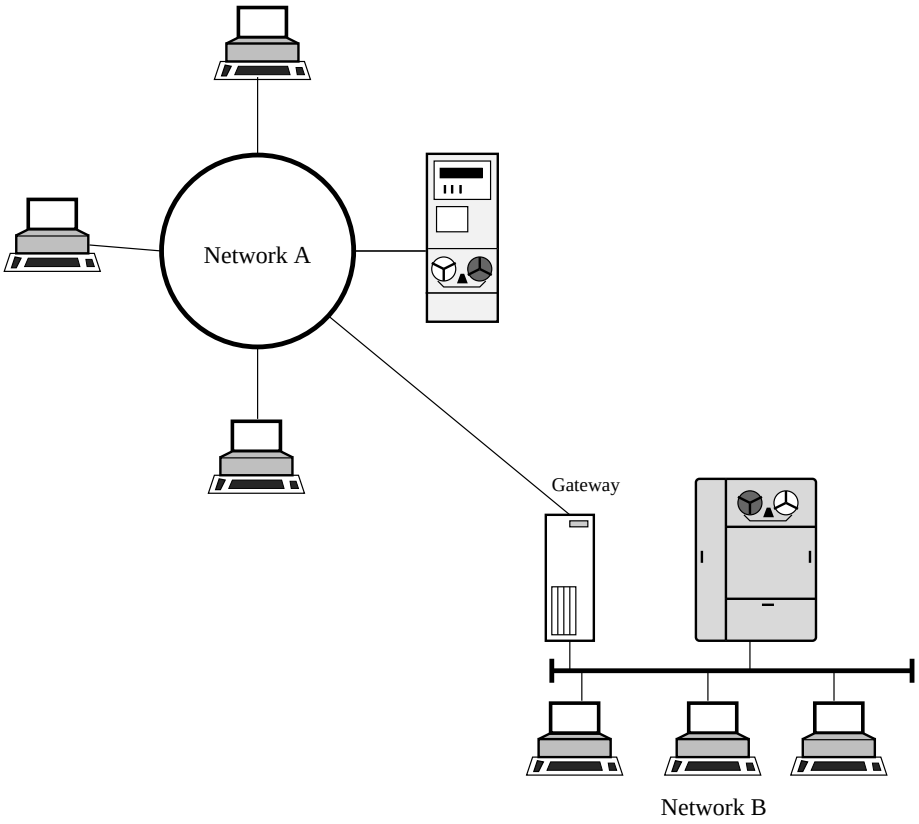

Figure 3.1: *A catenet comprised of two networks*

- *Routing.* The destination host is not necessarily on the local physical network. IP is responsible for routing the datagram through the gateways to the destination host.
- *Status communication.* IP reports the upper layers protocol of the status of outgoing datagrams. Events reported include the destination being unreachable, datagram time-out, etc.

IP employs a best effort delivery scheme. This means the IP layer attempts delivering the datagram to the destination host, but it does not guarantee that the datagram is, eventually, delivered, or even that a delivery failure is reported to the user.

IP datagrams may be lost, duplicated, arrive out of sequence, arrive with errors, or any combination of the above. It only guarantee that a delivered datagram is delivered to the right destination host, and is transferred to the transport protocol it belongs to. IP does not support other networking issues such as flow control, and congestion control. The support of these services is left for the upper layer protocols.

3.1.2 UDP

The User Datagram Protocol [Pos80] is a simple, unreliable, connectionless, transport protocol. UDP enhances the services provided by IP by multiplexing and demultiplexing datagrams in the source and destination hosts, respectively, and optionally adds some reliability by detecting errors in delivered datagrams.

UDP defines, in each host, a set of ports. Ports are abstract destination points identified by a positive integer. Datagrams arriving at a host are demultiplexed according to the destination port number. Each UDP datagram includes the source port number, which specifies the port number to which replies should be sent.

Most implementation of UDP support the notion of a UDP connection. A UDP connection is defined by the host addresses, and the port numbers, of the communicating entities. The semantics of a UDP connection are that each entity can send datagrams to the other entity only, and that datagrams sent from other entities to either sides of the connection are discarded. No handshaking, or reliability mechanisms are supported by the connection.

Error detection is done by adding a 16 bit checksum to each outgoing datagram. This checksum is checked in incoming datagrams, and if an error is detected the datagram is discarded. There is no need to report the source host of the discarded datagram because datagram delivery is not guaranteed by UDP.

3.1.3 TCP

The Transmission Control Protocol [Pos81b] is the main transport layer protocol provided by the Internet Protocols suite. TCP provides for connections of bidirectional reliable transfer of streams of data. It also handles congestion and flow control.

TCP, like UDP, defines a set of ports in each host. A TCP connection is defined by the host addresses and the port numbers on both sides of the connection. The connection is handled as two unidirectional connections, with each side acting as both sender and receiver. Control data of a receiver side may be, however, piggybacked on outgoing data packets.

TCP is a sliding window protocol. The sender keeps a transmit window which slides across the data stream, and may only send data from within the window. When the receiver receives in-sequence data it acknowledges the sender of the data receipt, the sender transmit window is moved to after the acknowledged data, and new data can be sent. The receiver keeps a receive

window which is identical to the transmit window of the sender. By disposing an incoming data that does not fit in the window or already exists in it, the receiver is able to overcome duplicate messages. An out of sequence data is inserted to the receive window, but is not acknowledged until all the preceding data is received.

Data loss is handled by retransmission. When a data byte is sent a timer is started. If the timer expires before the data is acknowledged, the data is retransmitted, and the timer is started again. To avoid congestion due to retransmissions, TCP employs an exponential backoff timer strategy. The value of the retransmission timers grows exponentially with the number of retransmissions. This prevents possible congestion in the network by reducing the network traffic volume.

The window size used by the sender is not constant. The receiver notifies the sender of the available buffer space in the receiver, and the sender uses this value as a limit to the window size. This method provides flow control to TCP.

A TCP connection is established using a three way handshake protocol. During connection establishment both parties exchange some information required for connection management. This information includes the receive window size, the sequence number of the first data byte, and the maximum size of each TCP segment.

A segment is the term by which TCP refers to the data sent in one IP datagram. Since the probability of a datagram loss grows with the number of transmitted fragments, TCP attempts at passing IP datagrams that need no fragmentation. This is done by each side informing the other what is the maximum segment size his local network can handle with no fragmentation. The maximum segment size used by a connection is the minimum of these two values.

Most TCP implementations implement the Nagle algorithm [Nag84] to coalesce short segments. A sender employing this algorithm buffers outgoing data when it expects some more data to be sent. Generally if there is outstanding unacknowledged data, the sending TCP buffers data until the outstanding data is acknowledged, or it can send a full segment.

3.1.4 RDP

The Reliable Data Protocol [VHS84, PaH90] is an experimental internet protocol which supports reliable datagram-based connections.

RDP is a sliding window protocol. As the basic data unit in RDP is a datagram, the window size is defined by the number of datagrams the sender can transmit. Unlike TCP, RDP allows two

types of acknowledgment. A cumulative acknowledgment is used to acknowledge all datagrams up to a specified one. A selective acknowledgment can be used by the receiver to acknowledge datagrams arriving out of sequence. (TCP does not support this acknowledgment type.)

The advantages of RDP over TCP are its simplicity, the fact that it preserves message boundaries, and its ability to transfer datagrams at the order they arrived, and not the order they were sent.

3.1.5 RPC

Unlike the aforementioned protocols, the Remote Procedure Call [Sun88] is not a transport layer protocol. It correspond to the session layer of the OSI reference model. The concept of a remote procedure call is based on the client-server model of communication. In this model one of the communicating entities—the server—provides services for other entities—the clients. These services vary from server to server. They can be as simple as reporting the current time, or as complicated as providing access to a global database.

The remote procedure call communication model is a type of client-server communication model, in which the server executes a procedure in behalf of the client. RPC defines the format of the client request, and the server reply. It does not deal with the nature of the underlying transport protocol. If the underlying protocol is not reliable RPC adds reliability by request retransmission. Each request is identified by a unique request ID. This ID enables the server to detect duplicate messages, and the client to match the reply with a request.

3.2 Intraprocess communication

When choosing the transport protocol to be used for the communication between the deputy and the body we had to consider several requirements. For the execution of system calls the protocol must be reliable. As the protocol will usually be used for transfer of many small packets, it should also have short latency. Since the network is a common resource, the protocol should have as small network overhead as possible, and as the network might be used for transferring large amounts of data the protocol must also be able to handle congestion.

This section presents the considerations used in selecting the protocols for the intraprocess communication, and modifications applied to the transport protocol.

3.2.1 The Intraprocess Transport Protocol

Of the three transport layer protocols described above the choice, naturally falls on TCP. Raw UDP does not satisfy the requirements because it is inherently unreliable. Adding reliability in the intraprocess protocol is most likely to be less efficient then existing protocols, and is very likely to induce problems due to incorrect design or implementation.

RDP might have been suitable for the requirements had it supported any mechanism for congestion handling. The lack of congestion control is made more critical by the fact that RDP relies on IP to fragment and reassemble datagrams. If any of the fragments is lost the rest are kept in the destination machine until they are timed out, and the datagram is lost. A lost fragment will, therefore, hold resources for the remaining fragments in the destination machine, thus worsening the congestion.

When a datagram is lost RDP has to retransmit the complete datagram again. The probability of datagram loss is almost proportional to the number of fragments composing the datagram.[1] Sending large datagrams is, therefore, more likely to cause datagram loss, and thus retransmission of the complete datagram. The probabilities of TCP segment loss are of course, not smaller, but TCP has only to send the lost segment, and not the complete datagram, thus reducing the volume of data sent over the network, and avoiding congestion even more.

3.2.2 The Intraprocess Protocol

The choice of the intraprocess protocol is simpler. The only standard candidate is RPC, which does not suit our needs. RPC is using XDR for data representation. The translation between host representation and XDR representation incurs heavy overhead on each transmitted packet.

The remote procedure call model fits the remote system call execution semantics. It can also accommodate the semantics of the deputy access to the body's memory during a system call by declaring the body as a server for the duration of the system call. The relationship between the deputy and the body cannot, however, be described by the standard client-server model, as either might initiate request to the other. The remote procedure call model can, therefore, never describe this relationship, and RPC is not a suitable communication protocol.

Since the standard protocol does not satisfy the requirements, a dedicated protocol was de-

[1] The exact probability is $1-(1-P)^n$, where P is the probability of a packet loss, and n is the number of fragments. For the typical values of $P = 10^{-3}$, and $n = 6$ the probability is 0.00598503.

veloped. This protocol is not a general protocol like RPC, but provides ad hoc solutions for the problem at hand. As the protocol is tightly connected to the implementation it is described in Chapter 4.

3.2.3 TCP Modifications

The current transport protocol used for the intraprocess communication is a modified version of TCP. Using some tuning and minor modifications we were able to improve the system performance considerably. The modifications, and their rationale are described here.

First, and most important, we inhibited the Nagle algorithm. This is supported by the system, but is not a default option. This option was disabled because the intraprocess protocol sometimes sends two consecutive replies. With outgoing data coalescence the second reply is delayed for an average period of 100ms, which is far too long for a system call.

Another change of the defaults was in enabling the keep-alive option, and modifying its parameters to close connections that are idle for more than 30 seconds. This option is not necessary for the system's operation, but it enables faster garbage collection in the presence of hosts or network crashes.

The last modification is the removal of the TCP checksum. TCP provides error detection by adding a checksum to each segment transmitted. This checksum is required in the standard internet environment, as some of the networks comprising it may incur errors. All modern networks provide error checking, and guarantee that delivered packets are delivered correctly. Hence, the TCP checksum in such an environment is redundant. The computation of this checksum, and its validation amounts to a significant part of the processing required by the protocol. Removing this checksum reduced latency by 20%. We believe this gain justifies the deviation from the standard.

Chapter 4

Implementation

This chapter presents the implementation of the deputy mechanism. The mechanism is implemented on top of the Berkeley Software Design, Inc. BSD/386 system. BSD/386 is a system derived from the 4.3BSD Network Release 2 operating system, developed in UC Berkeley.

The first two sections present the structures of the process body and the deputy. The interaction between the two is described in the third section. Some special cases are presented in the fourth section.

4.1 The Body

Previous chapters stated that the body is the user context of a process. This is basically true, but as a process in UNIX must have a system context for basic system mechanisms to operate on, the body is implemented as a complete UNIX process.

Each body has a communication channel associated with it. Data transmitted on this channel arrives to the deputy, and data transmitted from the deputy arrives to the body's channel.

When the user code of the process executes a system call, the kernel checks if the process is a local process, or a body of a process from a remote PE. In the first case the standard dispatch table is used to handle the system call. In the second case a second dispatch table is consulted. Values at this second dispatch table can be of three types: local, remote, and special. Local system calls are executed at the PE which hosts the body of the process. Remote system calls are simply forwarded to the PE which has the deputy, and are executed there. Special system calls are those system calls that require special action to be taken. Of the 146 system calls supported by the BSD/386, 9 are

local, 133 are remote, and only 4 are special.

During the execution of the system call, the deputy might have to access the user address space of the process. These cases are handled by requests sent from the deputy, and served by the body. This mechanism is also used by the deputy for other requests, e.g. setting up memory when running a new program using the *execve()* system call.

Most of the body's memory management is handled at the host PE. The only exception to this are page in requests from the file system. This exception is essential, as file systems differ in different PEs. Paging from a file occurs frequently in pure executables, where the file is mapped to the process's memory, and is paged in on demand. Paging from a file will also occur as a result of the *mmap()* system call which is used to map files to the process's address space. This paging is handled by sending the deputy a request for data. The deputy then reads the data and sends it to the body.

The body, like a standard process, checks for signals before the transit to user mode. Signals are, however, treated in a different way. Synchronous signals, such as floating point overflow, or segmentation violation, which are generated by the PE in which the body is executing, are forwarded to the deputy. The deputy is responsible for processing the signals, deciding what action to take, and informing the body of this action. When checking for signals, the body also checks if the deputy had requested its attention, and if so it sends a null request to the deputy.

4.2 The Deputy

The deputy is implemented as a UNIX process. The deputy, unlike regular processes, has only system context, and no user context. Like the body, the deputy has a communication channel, which is used to communicate with the body.

Generally, the deputy is in a wait state, waiting for the occurance of an event. When an event occurs, the deputy handles it, and when it finishes it returns to the wait state. An event can be either a request from it's body, or the posting of a signal to the process. In case of an asynchronous event, e.g. the posting of a signal, the deputy sends the body a request to initiate a communication transaction. This request is required to avoid race conditions that might result from the asynchronous nature of the event.

While serving the request the deputy might have to access the user's address space. Access attempts should be intercepted and sent to the body. To enable this, the interface from the system

context to the user memory is modified. Functions like *copyin()* and *copyout()* first check if the running process is a deputy. If it is not, the standard function is executed. Otherwise a request for data transfer is sent to the body, which handles the transfer.

4.3 Interaction Management

The body and the deputy communicate over a communication channel using a modified version of TCP as the transport protocol. The rationale for using TCP and the modification applied, are explained in Chapter 3.

The interaction between the body and the deputy is done in transactions. A transaction starts when the body asks the deputy to perform a task on its behalf, and ends after the deputy handles the request and reports the body of all asynchronous events that had occurred since the previous transaction. This way the body is always the initiator of transactions, and the deputy always terminates it.

For the execution of a system call, the body sends the system call number and arguments to the deputy. For simple system calls, such as *socket()* or *close()*, the deputy executes the system call, and returns its result. Then, if no asynchronous events are pending, the deputy informs the body that the transaction terminates. Figure 4.1 (A) demonstrates the flow control during the execution of the socket system call. If no asynchronous events are pending the transaction termination message is piggybacked on the system call result message.

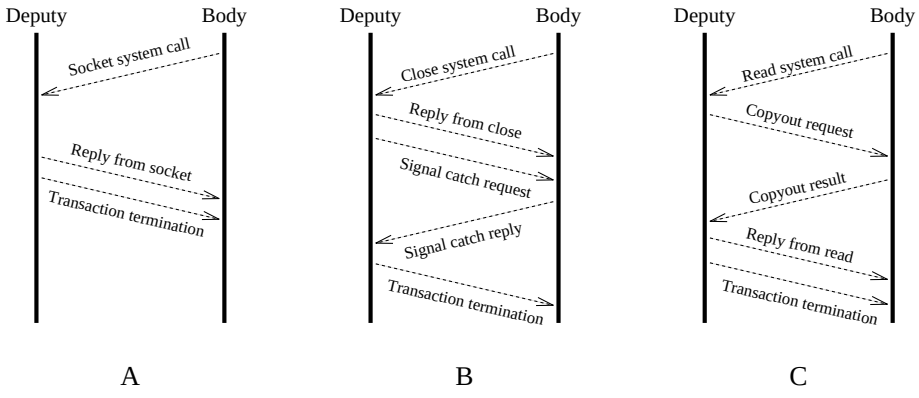

Figure 4.1: *Communication flow in various system calls*

If there are asynchronous events pending, the deputy requests the body to take the appropriate action before terminating the transaction. This situation is demonstrated in Figure 4.1 (B), where

the process receives a signal it arranged to catch during the execution of a close system call. In this case the system call immediately terminates, and the body is requested to arrange for the execution of the signal handler.

As already mentioned, in some system calls the deputy might access the user data space. Figure 4.1 (C) shows the flow control in the system call *read()*, when the deputy executes one copyout request.

When a signal is sent at the process, the deputy has to inform the process of the action to be taken. If the deputy is in the middle of serving a body's request, it has nothing to do until it is done serving, when the action will be sent to the process. If, however, the deputy is not serving a request, it sends the body a request to initiate a transaction. The body will respond by sending a null request. This null request will initiate a transaction, and the deputy will inform the body of the action to be taken before it terminates the transaction. The sequence of operations taken in this case is shown in Figure 4.2 (A).

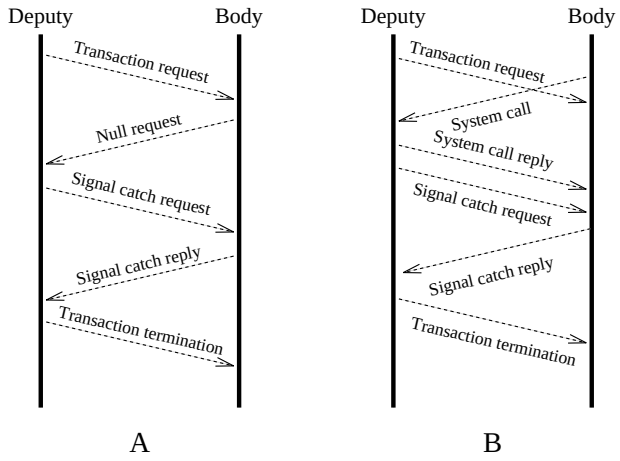

Figure 4.2: *Communication flow in signal handling*

The first two messages may seem to be redundant; it may seem that it is possible for the deputy to initiate a transaction, and send the action as the first message. The problem with this scheme is that it may cause a race condition. If the body has sent a system call request, which have not yet arrived, or will attempt to send a request before it receives the action, both sides will have to agree on which event occurred first. To preserve the UNIX [LMKQ89] semantics, this action should always be the system call. The deputy will then have to roll-back all of the signal processing it did, serve the system call, and process the signal again. The body cannot keep the signal action, and executes it after the system call since the system call might block the signal, or modify the

action to be taken. Eliminating the transaction and null request messages will result in greater complexity in the deputy and, as signals are not very frequent, seems to be unnecessary.

If the same race condition occurs between the transaction request message and a system call, the body simply ignores the transaction request. The deputy is waiting for a request from the body, and will serve a system call if it arrives after sending the transaction request. When the system call is done, the deputy will process the signal, and send the action to the body. Figure 4.2 (B) is an example of the flow in this case.

4.4 Special Cases

This section presents two special cases; the *fork()* system call, which creates a new process, and the *migrate()* system call, which is a new system call that moves the process to another machine.

4.4.1 Fork

In BSD/386, like UNIX, the only way for a user to create a new process is to invoke the *fork()* system call. The invoking process is called the *parent* process, and the newly created process is called the *child* process. On return from the system call the child process is an exact duplicate of the parent. The two differ only in the process ID.

When a process whose body have migrated forks both deputy and body are forked, and a communication channel is established between the two parts of the child process. Figure 4.3 shows some of the stages in accomplishing this. We start from one split process (A). The body sends a request to the deputy to open a communication channel, and forks a new deputy (B). The body then forks, leaving the new communication channel for the child body.

As the deputy forks in the middle of the transaction, both parent and child deputies send the transaction termination message. The parent deputy forks after it receives the termination signal. When forking the body, care should be taken that the child waits for a termination message on his channel.

4.4.2 Migration

The system supports a new system call that moves the body to a new PE. Migrating the process involves several stages. These stages are depicted in Figure 4.4.

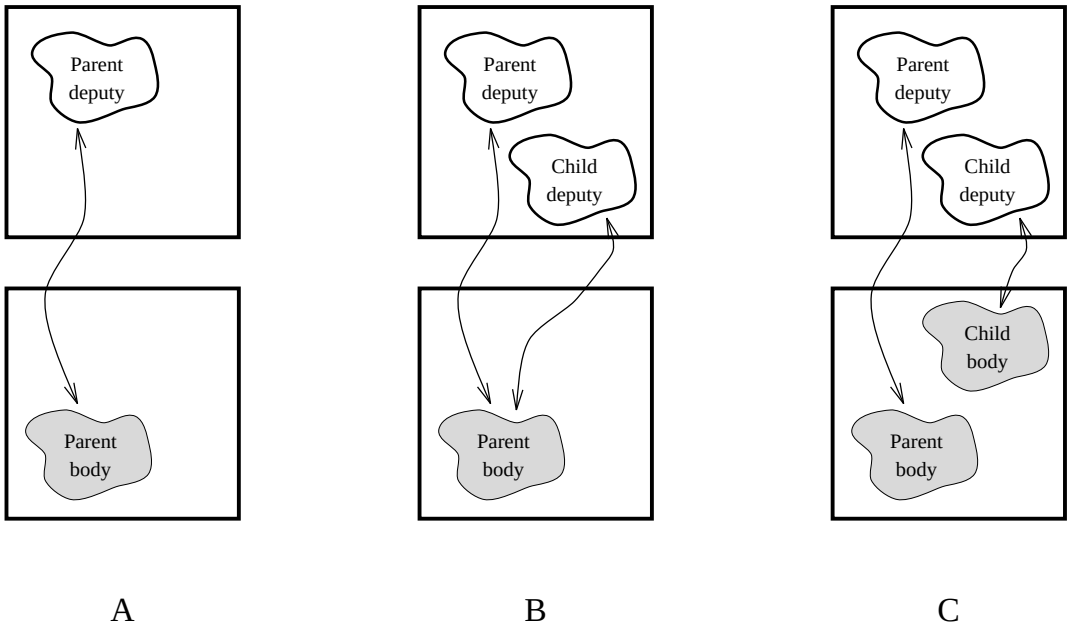

Figure 4.3: *The* fork() *system call*

In the initial state (A), we have a split process. When the body invokes the migrate system call, the deputy starts a process on the target PE, which will be the new body, and holds a communication channel to it (B). The first communication to the target PE is done using a migration daemon on it, which waits for migration attempts, and creates the processes for the new bodies.

The new body uses its indirect channel via the deputy, to inform the body of the address of a communication port it opens for the actual migration. The body connects to this address, establishing a third communication link (C). Using this new communication link, the body sends its state (i.e. memory maps, memory contents, mmaped files information, program state, etc.) to the new body, which becomes an exact duplicate of the body. The original body then closes the connection down, and terminates, leaving the new body in the target PE (D).

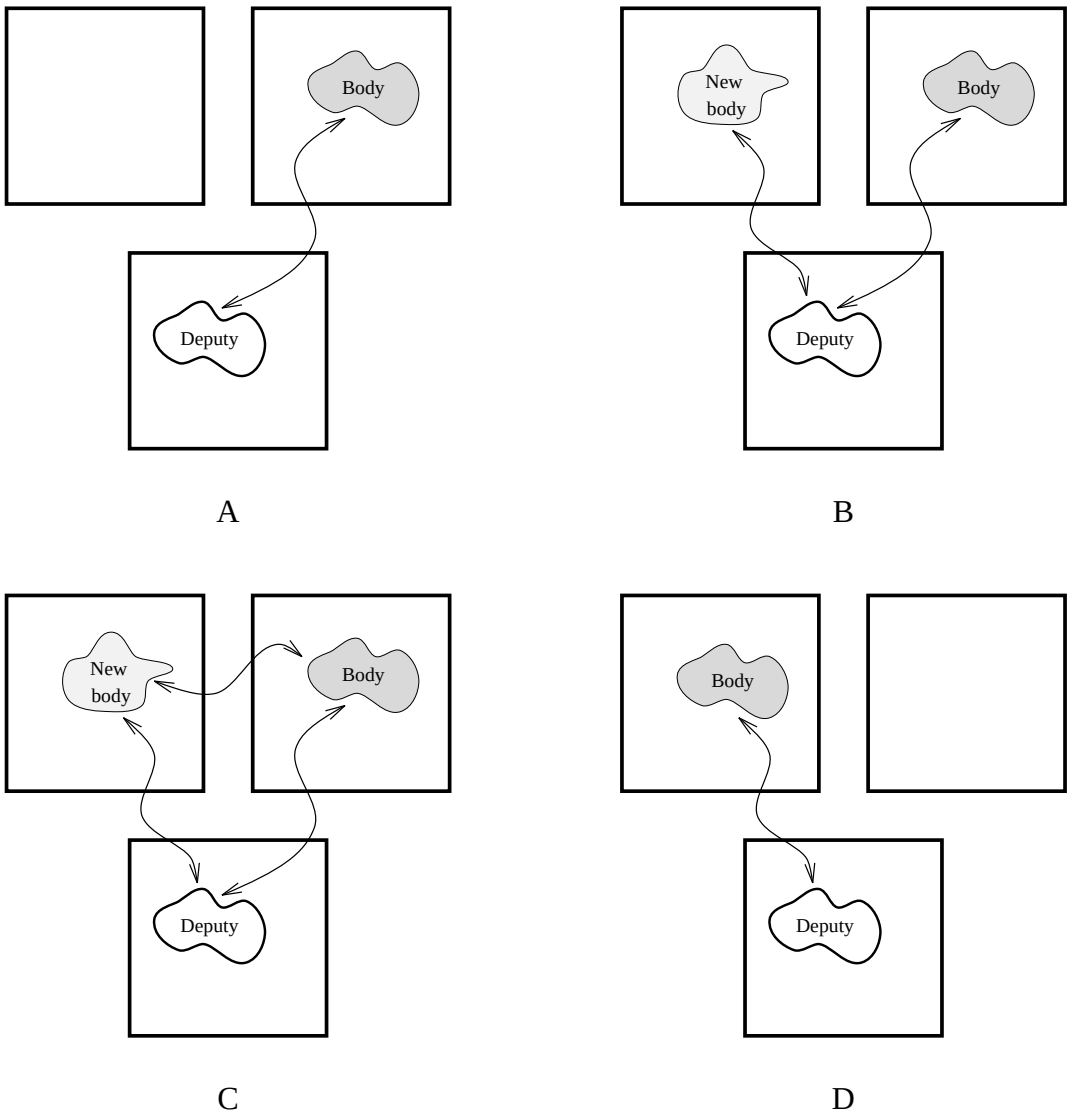

Figure 4.4: *Stages in process migration*

Chapter 5

Performance

This chapter presents performance results obtained from the implemented prototype. The tests were executed on 2 Intel Professional/GX workstations, each with an Intel 486DX/33 Processor, 16MB main memory, and an SMC WD8013EPC Ethernet controller.

The first section provides a comparison of remote vs. local system call execution time. Section 2 presents a dissection of the overhead in the execution of remote system calls.

5.1 Remote System Calls

System calls are the main interface between the deputy and the body. The frequent use of system calls in processes make them the key factor in the remote execution overhead. This section shows the overhead of executing a remote system call.

5.1.1 Measurement Technique

Remote system calls overhead was measured by running a set of benchmarking programs. Each benchmark measures the execution time of one system call. A system call is executed 100,000 times on the local PE, and 10,000 times on the remote PE. The repetition is required to disseminate the process initialization overhead, and to avoid fluctuations due to the clock precision. Increasing the number of repetitions has no significant effect on the results.

This measurement method is good for idempotent system calls, whose execution time does not depend on the number of previously executed system calls. It fails, however, for system calls that modify the state of the process, or the object they operate upon. For example, the *write()* system

call enlarges a file each time it is executed. To measure such system call execution times, the benchmark nullifies the side effects of the measured system call using an idempotent system call, the execution time of which is known. The execution time of the nullifying system call is then subtracted from that of the combined calls to get the execution time of the measured system call.

5.1.2 Results

Performance measurements were performed for the following system calls:

- *close()*
- *lseek()*
- *open()*
- *write()*

The measurements results are summarized in Table 5.1.

Syscall	Local (ms)	Remote (ms)	Slowdown
close	.024	3.27	136.3
lseek	.025	3.28	131.2
open	.610	7.43	12.2
write 1K	.340	9.84	28.9
write 8K	1.450	24.65	17.0

Table 5.1: *Execution times of remote vs. local system calls*

As expected, remote execution of short system calls has a very large slowdown ratio. Heavier system calls, which require data transfer, has a larger absolute overhead, but the slowdown ratio is much lower.

5.1.3 A Closer Look on *write()*

This section provides a closer look on the *write()* system call. In particular we show the effects of the size of the written block on the performance of the remote write. Figure 5.1 displays the remote and local execution times for the *write()* system call, with varying block sizes.

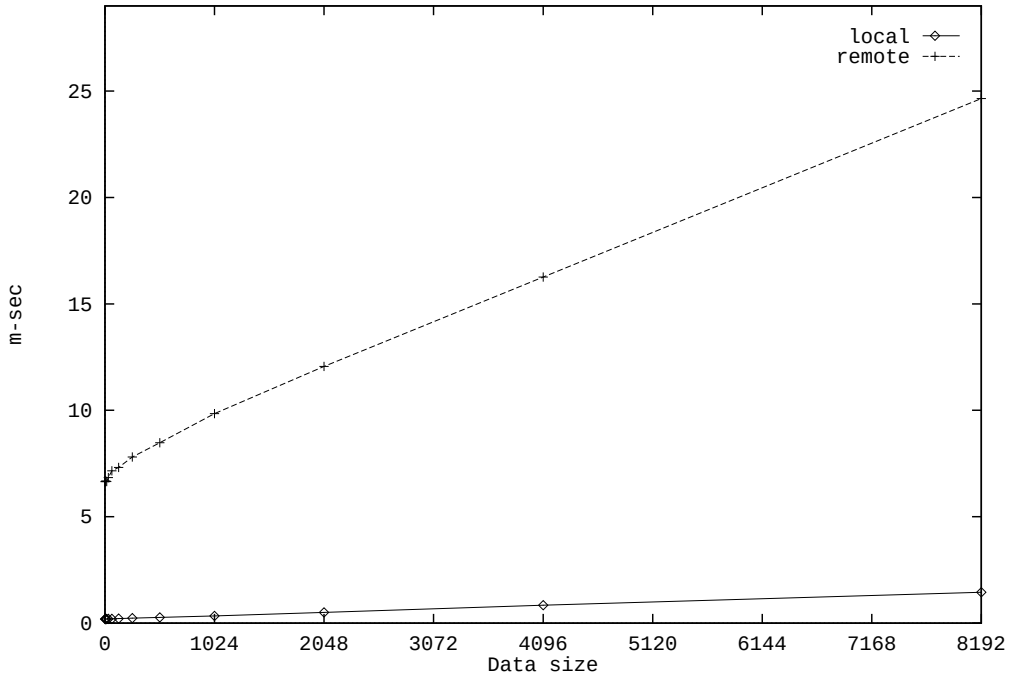

Figure 5.1: *Write() system call execution times*

As the graph shows, the execution times of both the local and the remote versions of the *write()* system call are proportional with the size of the written block. As the packet size increases, the slowdown ratio is decreased, as shown in Figure 5.2.

5.2 Overhead Dissection

The previous section presented the overhead incurred in remote execution vs. local execution of various system calls. In order to choose the right approach towards improving these ratio, one has to know the weight of the various components of the overhead.

This section presents a dissection of the overhead into its various components.

5.2.1 Factoring the Overhead

As the process uses the communication channel only to forward requests and replies to/from the deputy, the only source for overhead is the communication. The total overhead of a given system call is, therefore, the sum of the *message latencies* of the communication messages sent between the body and the deputy, where *message latency* is defined as the time since one side of the

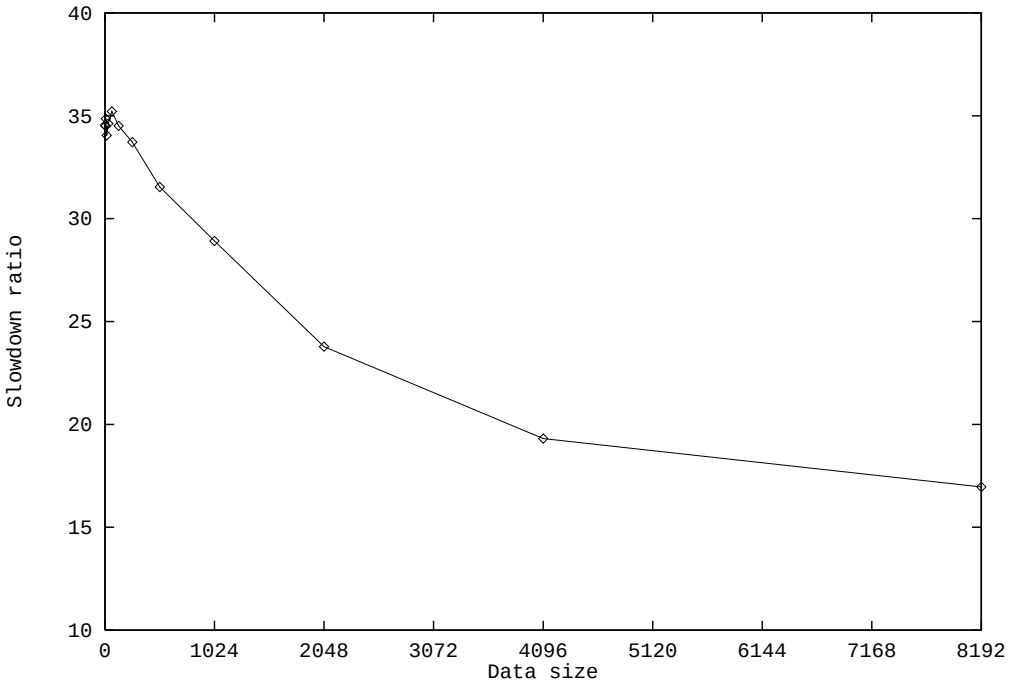

Figure 5.2: *Write() system call execution slowdown*

communication decides to send a message to the other, until the message is accepted by the other side.

The message latency has the following components:

- **Network Latency** This is the time since a packet is delivered to the interface at the sending machine, until the packet is collected from the network by the receiving machine's network interface, and is passed to the system to be handled.
- **IP Overhead** This is the overhead incurred by IP's handling of the packet, at both the sending and the receiving machines.
- **TCP overhead** The overhead induced by the TCP protocol management.
- **Data Packaging** The time it takes for the sending machine to package the data, and for the receiving machine to unpackage it.
- **Context Switch** The receiving process is usually not active. This part includes the time it takes to re-activate it.

The following sections describe these components, the methods used to measure each of them, and the measurements results.

Network Latency

The network latency is the time it takes for the network to transfer a message from one node to another. More precisely, this is the time since a packet is handed to the interface layer in the sending node, until the interface layer at the receiving node deliver the packet to the upper layers.

Measuring the network latency was done by writing a 'ping-pong' protocol. This protocol simply mirrors the first 10,000 packets it receives from the network interface, at which time the packets are dropped. By sending one packet to this protocol on the remote node, and monitoring the time it takes for the protocol to start dropping packets we were able to measure the round-trip time. As the protocol does nothing but forwarding the packet, the network latency is half the round-trip time. Measured latencies for various packet sizes on an Ethernet network are shown in Figure 5.3.

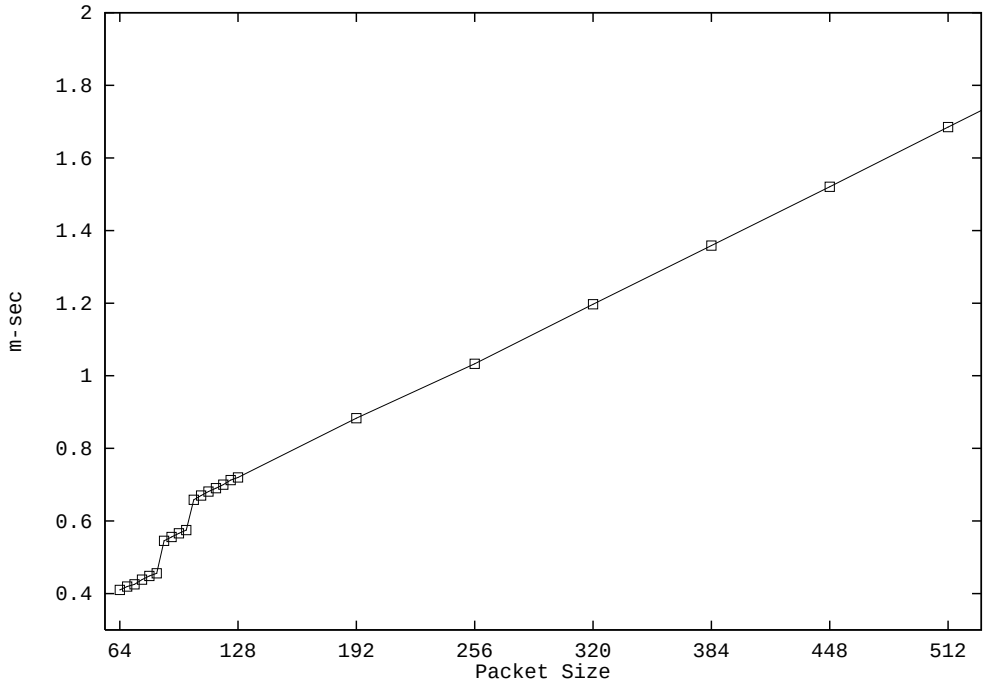

Figure 5.3: *Ethernet network latency (ms)*

The measured network latency is roughly $(328 + 2.59n)\mu$s for an n bytes packet. The network interface is optimized for small packets, and is able to reduce this figure by 41μs for packets of size

between 89 and 104 bytes, and 83μs for smaller packets.

IP Overhead

IP overhead was measured using the same method used for measuring the network latency, namely, a special purpose 'ping-pong' protocol. This method gives the round-trip time for an IP datagram. By subtracting the network latency from these results we were able to calculate the IP overhead. This overhead is 184μs, regardless of the packet size. Datagrams larger then an Ethernet packet were not measured, as TCP does not use the IP fragmentation.

Context Switch

To measure the context switch time the following benchmark was executed. A bidirectional communication channel was opened. One data byte was repeatedly sent from one endpoint to the other and back again. This benchmark was executed twice, the first time with one process doing all the operations, and in the second two processes were used, one at each communication endpoint. The only difference between the execution times of these two runs is due to the context switches required in the second run. The total execution time difference was divided by the number of context switches executed, and the result is an average of 165μs for a context switch.

The message latency includes one context switch at the receiving side. A context switch at the sending side will occur after the packet has been transmitted, and will therefore, overlap other events.

TCP Overhead

TCP was also measured using the same 'ping-pong' method, only the communicating entities were two user-level processes. By subtracting the IP overhead, network latency, context switch time, and system call overhead from the resulting round-trip time we found that the modified TCP (with no checksum) overhead is independent of the volume of transmitted data. This overhead amounts to 825μs.

Data Packaging

To measure the data packaging overhead, some of the benchmarks were executed on a special kernel that uses the packaging of the intraprocess communication, but instead of forwarding the

packets to/from a deputy, calls the unpackaging functions in the calling process. The resulting measurements gives an overhead of about $(75 + .03n)\mu$s for packaging n bytes.

5.2.2 Validating the Measurements

Validation of the above measurements was done by calculating the expected overhead for various system calls, and comparing the result with the measured overhead. Tables 5.2 and 5.3 present the calculation of the expected overhead for the system call *close()* and for the *write()* system call writing 1KB of data, respectively.

Factor	Request	Reply	Total
Network	411	421	832
IP	184	184	368
C-Switch	155	155	310
TCP	825	825	1,650
Packaging	77	77	154
Total	1,652	1,662	3,314

Table 5.2: *Expected close() system call overhead (μs)*

The overhead in the *close()* is due to two messages: the request, and the reply. The request is 64 bytes long, and the reply is 68 bytes long (including the 40 bytes headers of TCP and IP). The total expected overhead is 3.314ms and the measured overhead, as can be extracted from Table 5.1, is 3.256ms.

	System Call		*copyin()*		
Factor	Request	Reply	Request	Reply	Total
Network	431	421	390	3,125	4,367
IP	184	184	184	184	736
C-switch	155	155	155	155	620
TCP	825	825	825	825	3,300
Packaging	77	77	77	182	413
Total	1,672	1,662	1,631	4,471	9,436

Table 5.3: *Expected overhead of write() of 1KB (μs)*

The *write()* system call has a request size of 72 bytes, and the reply is 68 bytes long. During its execution *write()* calls *copyin()* to copy data from the body. The *copyin()* request is 56 bytes long. The reply, which contains the data, is 1,080 bytes long. The expected overhead of a remote invocation of a *write()* system call is 9.436ms, while the measured overhead is 9.5ms.

5.3 Communication Improvements

As can be seen in Figure 5.4, the network latency, and the TCP protocol overhead constitutes the major parts of the overhead for the measured cases. This section discusses the possible gains from improving those layers.

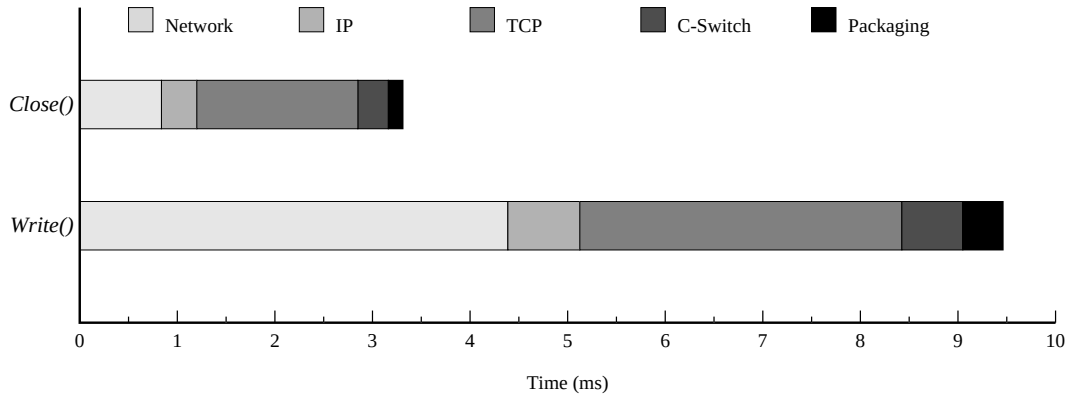

Figure 5.4: *Communication overhead components*

5.3.1 FDDI

The first considered improvement is replacing the Ethernet network used by the prototype to an FDDI network. FDDI is a token ring network with a bandwidth of 100Mbps, operating on fiber-optics technology. As the Ethernet's bandwidth is only 10Mbps, and since other features of the FDDI are designed to provide better performance, it is possible to assume that the FDDI performance is tenfold that of the Ethernet.

Given this assumption, the effect of replacing the Ethernet network by an FDDI will be reducing the network latency by 90%. This will reduce the overhead for a remote invocation of the *close()* system call to 2.565ms, or an improvement of 23%. The expected slowdown ratio is then 108. These results will be very similar for other light system calls, such as *lseek()* or *socket()*.

For system calls that require bulk data transfer, such as *write()*, the improvement will be

considerably better. For example, for the case of writing 1KB, the network latency will be reduced from 4.367ms to 437ms, which means the total overhead is reduced to 5.506ms, or an improvement of 43%.

5.3.2 Replacing TCP/IP

For short packets a major part of the overhead is caused by the TCP and IP protocols. The TCP protocol is designed to give acceptable performance for all types of networks. IP is designed to work in a dynamic, large internetworking environment. The generality of both protocols results in both large headers, and a large amount of computation.

We believe that a special-purpose, reliable protocol for a distributed environment can have a significant performance improvement over TCP/IP. If such a protocol can be implemented to be 10 times faster than TCP/IP, its use will considerably reduce the overhead incurred by a remote execution of a system call. The total overhead for the remote execution of the *close()* system call will be reduced to 1.498ms, or an improvement of 55%. For the *write()* system call the improvement will be 38%, and the overhead will be 5.804ms.

Chapter 6

Conclusions

This work presented the deputy mechanism for achieving transparent process migration. The major idea in the mechanism is splitting the process into two parts, the first of which does the actual computation, and the second is responsible for keeping the process environment. These two parts communicate across a communication channel.

We have implemented a prototype of a deputy-based distributed operating system. The prototype supports dynamic load balancing based on the migration mechanism presented in this work.

The implementation of the prototype required adding 7,200 lines of code to the original kernel sources, which are 190,000 lines long. 30 source files out of the original 560 were modified to insert hooks for the deputy, and body, and to break several routines. Of the aforementioned 7,200 lines, only 5% are machine dependent. These are those dealing with restoring the state of a migrating body, and with setting the floating point unit for the load balancing computations.

Further study is still required in the communication domain. The intraprocess communication protocol can be considerably improved, the choice of TCP as the transport protocol should be reconsidered, and the suitable hardware layer should be selected. A second area of research is singling out the frequently used resources, and making them globally available.

One possible family of such system calls is those dealing with interprocess communication. In the current prototype, data transferred between two processes might have to travel twice over the network. First from the sending body to its deputy, and then from the receiving process's deputy to its body. One of these hops can be avoided by migrating sockets (communication endpoints) with one of the processes that uses them. Implementing migrateable sockets will require a new transport layer protocol that can support reliable many-to-many communication.

Bibliography

[Bac86] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ (1986).

[BLL92] A. Bricker, M. Litzkow, and M. Livny, "Condor Technical Summary," Technical Report TR1069, University of Wisconsin - Madison, (January 1992).

[BGW93] A. Barak, S. Guday, and R. Wheeler, *The MOSIX Distributed Operating System, Load Balancing for UNIX*, Lecture Notes in Computer Science, Vol. 672, Springer-Verlag, NY, (1993)

[Cer78] V. Cerf, "The Catenet Model for Internetworking," Technical Report IEN 48, SRI Network Information Center, Menlo Park, CA (July 1978).

[Com91] D. E. Comer, *Internetworking With TCP/IP Vol. I: Principles, Protocols, and Architecture, 2nd ed.*, Prentice-Hall, Englewood Cliffs, NJ (1991).

[Dou89] F. Douglis, "Experience with Process Migration in Sprite," *Proceedings of the USENIX Distributed & Multiprocessor Systems Workshop*, (october 1989)

[LiS92] M. Litzkow, M. Solomon, "Supporting Checkpointing and Process Migration outside the UNIX Kernel," *Proceedings of the Winter USENIX conference*, pp. 283–290, (January 1992).

[LMKQ89] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA (1989)

[MKD92] P. E. McKenney and K. F. Dove, "Efficient Demultiplexing of Incoming TCP Packets," *Computing Systems*, 5(2) pp. 141–158, (Spring 1992).

[Nag84] J. Nagle, "Congestion Control in IP/TCP," RFC 896, SRI Network Information Center, Menlo Park, CA (January 1984).

[MRT90] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba A distributed Operating System for the 1990s," *IEEE Computer*, 23(5), (May 1990).

[PaH90] C. Partridge and R. Hinden, "Version 2 of the Reliable Data Protocol (RDP)," RFC 1151, SRI Network Information Center, Menlo Park, CA (April 1990).

[OCD88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *IEEE Computer*, 21(2), (February 1988).

[Pos80] J. Postel, "User Datagram Protocol," RFC 768, SRI Network Information Center, Menlo Park, CA (August 1980).

[Pos81a] J. Postel, "Internet Protocol," RFC 791, SRI Network Information Center, Menlo Park, CA (September 1981).

[Pos81b] J. Postel, "Transmission Control Protocol," RFC 793, SRI Network Information Center, Menlo Park, CA (September 1981).

[Sun87] Sun Microsystems, "XDR: External Data Representation Standard," RFC 1014, SRI Network Information Center, Menlo Park, CA (June 1987).

[Sun88] Sun Microsystems, "RPC: Remote Procedure Call Protocol Specification Version 2," RFC 1057, SRI Network Information Center, Menlo Park, CA (June 1988).

[VHS84] D. Velten, R. Hinden, and J. Sax, "Reliable Data Protocol," RFC 908, SRI Network Information Center, Menlo Park, CA (July 1984).