EXTENDING FLUKE IPC FOR TRANSPARENT REMOTE COMMUNICATION

by

Linus Peter Kamb

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

December 1998

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Linus Peter Kamb

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: John Carter

Wilson Hsieh

Jay Lepreau

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Linus Peter Kamb in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

John Carter
Chair, Supervisory Committee

Approved for the Major Department

Robert R. Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Distributed systems such as client-server applications and cluster-based parallel computation are an important part of modern computing. Distributed computing allows the balancing of processing load, increases program modularity, isolates functionality, and can provide an element of fault tolerance. In these environments, systems must be able to synchronize and share data through some mechanism for remote interprocess communication (IPC). Although distributed systems have many advantages, they also pose several challenges. One important challenge is transparency. It is desirable that applications can be written to a communication interface that hides the details of distribution.

One way to achieve transparency is through the extension of local communication mechanisms over a network for remote communication. The ability to transparently extend local communication depends on the semantics of the local IPC mechanisms. Unfortunately, those semantics are often driven by other architectural goals of the system and may not necessarily be best suited for remote communication.

This work describes a remote IPC implementation for the Fluke operating system and an analysis of the Fluke architecture, IPC system, and IPC semantics, with regard to the extension of local IPC for transparent remote communication. It shows that the overall complexity of both the kernel IPC subsystem and the network IPC implementation is considerably less than similar operating systems' IPC mechanisms, and that the Fluke IPC architecture is generally well-suited for transparent remote IPC. However, it also shows that the lack of kernel-provided reference counting caused more problems than it solved, and that the generality of an important Fluke kernel object, the "reference," makes it impossible for the network IPC system to provide completely transparent remote IPC without extensive additional services.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my committee chair, John Carter, for his support and direction throughout my graduate studies.

I would also like to thank the members of my committee, Jay Lepreau and Wilson Hsieh. Jay, for his enthusiasm and direction; and Wilson, for not letting me get away with too much.

I would also like to thank the members of the Fluke team, especially Mike Hibler, Bryan Ford, Pat Tullmann, and Godmar Back, for their patience and assistance.

I would especially like to thank my brother, Alexander Kamb, for his generous support and encouragement during my graduate studies.

CHAPTER 1

INTRODUCTION

Remote communication is a fundamental element of distributed systems. In operating systems created from scratch to support distribution, such as Amoeba and V [2, 3], the remote communication mechanism is a cornerstone of their development. Other systems have been extended to support remote communication, either explicitly as with sockets in Unix, or by transparent extension of the local interprocess communication (IPC) mechanism, as in Mach [4]. With the development and increasing importance of clustered and distributed computing, much attention has been paid to the issues and design of remote communication mechanisms.

The design of an operating system for a distributed environment often forces tradeoffs between what is optimal and desirable on a local node versus what is possible and efficient for the distributed system. Depending on the goals of the system, tradeoffs may be made in many different subsystems, including virtual memory, scheduling, and the IPC system.

The resolution of these tradeoffs must be carefully considered as to their effects on the system as a whole, both on the local node as well as in the distributed environment. The system should be structured to support distribution without overly compromising the performance in the local case. In a system designed to support transparent distribution, the decisions regarding the IPC system merit special consideration. The design and semantics of the IPC system must be evaluated with respect to how effectively and efficiently they can be extended over a network. Designing and building an operating system from scratch provides an excellent opportunity to evaluate these design decisions and their effects on the distributed system.

This work evaluates how the design and semantics of local IPC mechanisms extend over a network, as remote IPC, to support a distributed environment. Specifically, it considers which aspects of an IPC system extend gracefully in remote IPC and which lead to greater cost and complexity in the network IPC implementation.

The analysis is at two levels. At the lower level, the specifics of the IPC system implementation are evaluated as to their suitability for extension over a network to support a distributed environment. At this level, details such as the connection model and the naming mechanism are evaluated by considering the complexity added to support their implementation in the network IPC system.

At a higher level, system architecture, goals, and design decisions are evaluated in terms of their effect on the IPC system. Decisions made to support specific architectural features have both direct and indirect consequences with respect to IPC. For instance, the choice to support certain process organizations can impact the design of the IPC system. Where possible, these design decisions and goals are compared with similar aspects of other operating systems and their remote communication models.

In support of this evaluation, a network IPC system was designed and implemented for the *Fluke* operating system. The network IPC system, or netipc, provides a practical basis for the evaluation of IPC semantics and low-level mechanisms, as well as an important service module for the operating system.

The Fluke microkernel architecture was established prior to the design and implementation of the netipc system. The goal of this work is to evaluate, through the implementation of a remote IPC system, how the architecture and design choices made for Fluke extend in a distributed environment to support remote IPC.

1.1 The Fluke Operating System

The Fluke operating system is designed to provide an efficient and flexible microkernel-based environment that supports a "nested process" architecture and transparent distribution. In the nested process architecture, each process believes it has its own machine, similar to the virtual machine concept, but differs in that parent processes only provide and control those resources that they choose for the child nested process. Other resources are provided by processes higher in the "nesting" hierarchy and are simply passed through the parent [1]. With transparent distribution, a process can run unchanged independent of its location. A given process will also be unaware of its location relative to other related processes in its environment. These other related processes may be located on the same node or across the network.

Among other goals, the Fluke IPC system was designed to support the nested process architecture and enable transparent remote communication. It has several interesting and novel features, including persistent thread-to-thread connections and a set of three different "flavors" of messaging primitives with different communication semantics. It also supports transparent interposition on communication channels. Using interposition, a process can operate as an intermediary between a client and server with neither the client nor the server being aware of the intermediary's interposition on their communication.

An example of such a system could be a security monitor managing file accesses by untrusted clients. Trusted applications would issue file requests directly to the file server, whereas untrusted applications would have their requests screened by the security monitor. The untrusted client application would make requests as if directly to the file server; the security monitor would interpose on this communication channel and evaluate each request, passing on only those requests deemed appropriate. This arrangement frees the file server from having to be aware of security issues and allows trusted applications to operate with a file server unencumbered by security checks. Additionally, a trusted application in one environment may be untrusted in another, yet run unchanged in both.

Fluke's transparent interposition is achieved by using a capability-based model [5] of communication. Fluke *ports* are the receive points for messages. *Port references* give the holder the *capability* to send a message to a given port. The capability model provides the *abstract naming* that is necessary for nested processes, as well as for location transparency in a distributed environment. An abstractly named object does not convey any information about itself, such as its location or owner, through its name.

In the nested process model, this naming abstraction is essential. A process only knows the state of the world relative to *itself*. That is, it has a parent, possibly children, and peers. There is no sense of absolute. Location transparency is the extension of abstract naming to a distributed environment. Processes are unaware of the machine location of the processes with which they communicate. With complete location transparency, both the communication mechanisms and their semantics are the same regardless of the location of the communicating entities. In this way, processes do not have to be concerned about making explicit remote communications to specific machines.

1.2 Fluke Issues for Remote Communication

Although the abstraction provided by capabilities is very useful in supporting this transparency, their use leads to several related and dependent issues. One issue is the permanence of objects. When can a process safely destroy an object and reclaim resources allocated for that object? This problem is essentially a garbage collection issue, and is often solved using a system-provided "reference count" of the number of capabilities, or references, to an object. The Fluke designers chose not to provide reference counting, in part because they felt that the benefits of reference counts were overshadowed by the anticipated complexity of providing accurate and efficient distributed reference counting. Whereas this relieved the netipc system from the burden of maintaining distributed reference counts, it caused other complications, such as the inability of the netipc system to manage transient port objects.

Another aspect of the capability model is how it is used in the system. Whereas capabilities are, in general, a mechanism to provide controlled access to an object or resource, most systems use them in conjunction with IPC. Although Fluke ports and *port* references are purely for IPC, Fluke *references* are general purpose objects that are used for more than just IPC. This particular aspect of the Fluke architecture had a significant impact on the netipc system. Fluke reference objects of all types can be passed through local IPC channels. Within the modular Fluke microkernel architecture, the network IPC system is expected to provide remote communication. It is not designed nor expected to be a mega-server that manages the distribution of all types of kernel objects. It therefore does not, and cannot, manage references to arbitrary objects such as references to memory regions. Because it cannot manage these references *on its own*, this aspect of the Fluke capability model limits the netipc system's ability to provide completely transparent remote IPC.

Whereas Fluke IPC is capability-based, using ports and references to initiate communication, the actual IPC connections are between two distinct threads. These connections can be long-lived, and must continue to be handled by the original pair of threads. For the netipc system, this meant that once a netipc system thread became involved in a specific remote connection, it had to continue to service that connection until it was disconnected. Incoming network packets must therefore be demultiplexed to specific threads. The current network interface in Fluke does not directly support such demultiplexing of packets, which is essentially a packet-filtering operation, so the netipc system must do its own internal demultiplexing causing additional synchronization and an extra context switch.

The Fluke IPC API provides distinct interfaces for its three different types of messaging. This mechanism was motivated by system architectural concerns as well as by optimization possibilities. Instead of describing the requested messaging behavior semantics through additional parameters and flags to a single IPC interface, Fluke defines a wider interface that allows each path through the kernel to be optimized and specialized for that particular behavior. This differentiation allows the code paths through the netipc system to be similarly optimized.

1.3 Thesis Summary

The Fluke IPC system was expected to extend gracefully and transparently in to a distributed environment. Several aspects do, indeed, support the extension of local IPC for remote communication, such as the use of ports and references.

Certain design features required some additional complexity and cost to support, such as the persistent thread-to-thread connections. Other features, including the distinct interfaces and Fluke's IPC buffer management, allowed a simpler and cleaner implementation of certain aspects of remote IPC. The capability model works well for the nested process architecture and the goal of providing remote communication through a user-level server. The general nature of the capability model is well-suited for transparent remote communication, but Fluke's particular implementation of capabilities limits completely transparent remote IPC.

The designers of Fluke chose to leave out some features found in other IPC systems because of the expected complexity-to-value tradeoff for local or remote IPC, or both. Some of these design decisions, such as choosing not to provide a mechanism by which to transfer a port to another process, and limiting the messaging semantics and guarantees, were clearly correct. Both the Fluke kernel IPC system and the network IPC implementation are considerably smaller and less complex than other, similar systems.

Other decisions are not as clearly advantageous. Kernel-supported reference counting was not included because of the anticipated added complexity of providing distributed reference counting. Without it, however, servers — and the netipc system in particular — cannot reliably manage their resources. Whereas an external service may reasonably provide reference-counting and notification services for processes, a performance-critical system such as the netipc system would be severely handicapped if required to contact an external server for reference counts.

As a whole, the design of Fluke and its IPC system is well-suited to the provision of remote IPC by a user-level server. The generality of Fluke's reference objects limit the network IPC system's ability to provide completely transparent remote IPC unaided, but other significant choices about the architecture and semantics of IPC enabled a simple and straightforward implementation of remote IPC.

1.4 Document Organization

The next chapter presents a brief overview of the Fluke operating system and its IPC model. Chapter 3 presents several other operating systems and their remote IPC models for comparison. Chapter 4 describes the design and implementation of the Fluke network IPC system. The presentation of the network IPC implementation is isolated from the analysis of the issues involved in the system's design to allow a clear and succinct presentation of the system. Chapter 5 then presents the evaluation of the Fluke IPC system and its semantics with respect to their extension over a network for remote IPC. Additionally, this chapter evaluates Fluke design decisions that affect the IPC system including comparisons to other operating systems' communication systems' mechanisms. Finally, Chapters 6 and 7 present some future research directions and the conclusion.

CHAPTER 2

THE FLUKE OPERATING SYSTEM

The Fluke operating system is a research operating system designed to investigate a novel architecture supporting "nested processes." In this architecture parent processes, called *nesters*, provide and control, as desired, all the resources given to their children processes. This architecture is similar to the virtual machine concept [6], but differs in important ways. Primarily, nesters do not need to support the entire virtual machine. They only provide and control for their children those resources that they choose to monitor.

Another design goal of the Fluke architecture is the support of transparent distribution. Certain key elements and features designed for the nested process model also lend themselves naturally to transparent distribution. For instance, an essential element of the nested process architecture is the use of capabilities to abstractly name objects. The abstraction provided by capabilities easily extends to help support location transparency in a distributed environment.

This chapter provides a brief overview of the Fluke architecture, certain key elements and design features, and a description of the Fluke IPC system. For a more complete discussion of Fluke, its design and motivation, and API, see [7, 8, 1].

2.1 Nesting and the Fluke Architecture

The primary architectural feature of the Fluke microkernel architecture is the nested process model. In this model, processes are "nested" within controlling parent processes. The child processes have direct access to non-supervisor machine instructions as well as to the Fluke kernel API. The kernel API provides the IPC interface as well as operations on the basic kernel objects, such as creating and manipulating ports, threads, and mutexes, etc. Controlled resources, such as files and processes, are provided through an IPC-based API called the "Common Protocols," discussed in section 2.1.3. These resources may be monitored by nesters through their selective interposition on the interfaces for those resources. This architecture is illustrated in Figure 2.1. Selective interposition confines the performance penalty that is traditionally associated with similar virtual machine or nested architectures to only those resources that are monitored.

2.1.1 Nesters

Nesters are a fundamental process type in the Fluke environment. A nester is a parent process that provides or monitors access to resources for its children. A parent does not "fork" off an independent child, as in a Unix environment. Rather, the child is created, and only exists, within the environment provided by the parent nester.

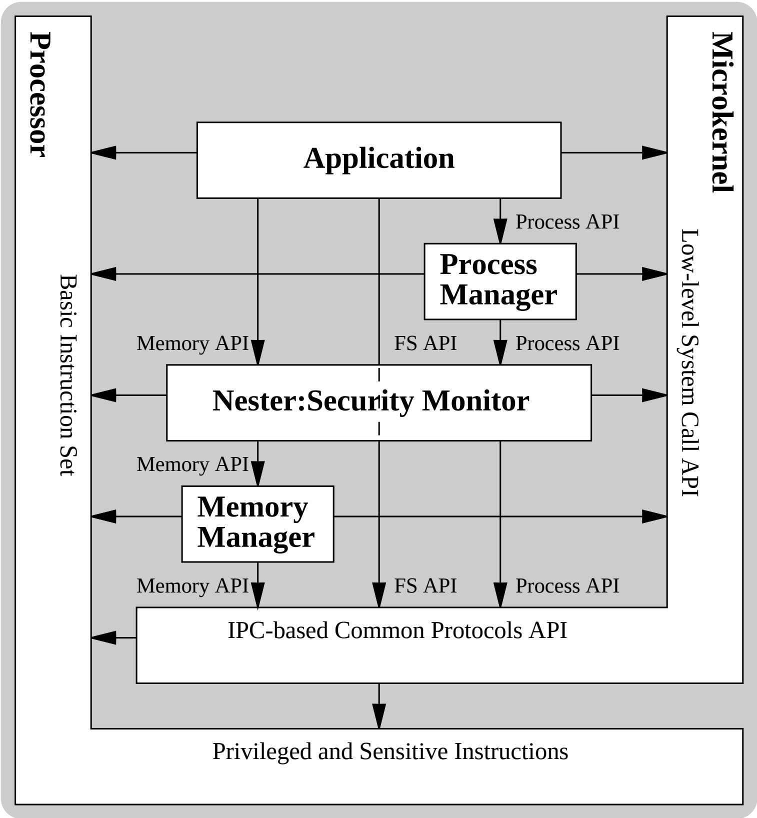

Figure 2.1. Illustration of the Fluke nested architecture. Adapted from [1].

The environment provided to the child by the nester includes access to key system resources, such as a memory server and the file system. The nester establishes itself as the provider of resources it is either providing directly or wishes to monitor. All of the child's requests for access to these resources will be handled, at least initially, by the nester. If the nester is only monitoring resource access, it may pass the request on to be serviced by its parent — the child's grandparent, which may also be a nester.

For access to resources about which the parent nester is not concerned, the nester simply passes on to its child that part of its own environment which was provided to it, in turn, by the parent's parent nester. The mechanisms nesters use to implement this functionality are described in greater detail later in this chapter.

2.1.2 Fluke Objects

All of the primary abstractions in Fluke are encapsulated in Fluke *objects*. Types of objects include address *spaces*, memory *regions*, *threads*, and IPC *ports*. The Fluke equivalent of a Unix *process* consists of one or more spaces and executing threads. An object consists of both reserved words in a process' address space and some kernel-internal state. The words in the process's address space are reserved for kernel use and must not be manipulated directly by the process.[1] Objects are manipulated by kernel calls to invoke operations on the object, which may involve access to the object's reserved words in the process' space, manipulation of kernel-internal state, or both.

Most objects can be "referenced." A Fluke *reference* is itself a kind of object. (References, however, cannot be referenced.) A reference to an object is a kind of pointer, or link, to the actual object. Although references are opaque, they may be queried for the type of the referenced object, and they can be compared for equality, i.e., that two references both refer to the same object. Beyond that, the

[1]This user-space part of an object is reserved for the kernel's use for possible optimizations. User manipulation of this state will either invalidate the possible optimization, or cause an exception for the process.

holder of a reference cannot determine anything more about the object referenced than is available through the object's public interface.

Many object operations, in fact, require a reference to the object. An example is in the IPC system. Port objects are the receive points for IPC messages. To send a message to a port, however, a process must possess a reference to that port. The IPC send operation takes as a parameter a reference to the port to which the message is destined.

2.1.3 Interfaces

Fluke defines several primary abstractions with its objects. Whereas these abstractions are intended to support the nested process model, they are also intended to allow the support of alternate architectures on top of Fluke. To complete the support of nested processes, there is a set of interfaces used by compliant processes.[2] These interfaces are known as the "Common Protocols" and include, among others, the file system interface and the memory interface. Supporting these interfaces are base servers, such as the Virtual Memory Manager for the memory interface. These interfaces are a separate part of the Fluke microkernel architecture, and an essential aspect of the nested process model.

One important Common Protocol is the "parent" interface. Nesters must minimally support this interface. The parent interface provides the universal "service service" for processes. Child processes use the parent interface to build their environment and to acquire references to key services, such as the file server. It is primarily by controlling this interface that nesters can control their children, as they provide the references to all other services the child requires. For instance, if the nester wished to monitor file accesses by its child, it would provide a reference to itself as the file server. The nester would then have to support the entire file system interface, though it would not have to support file operations. Accepted operations would be passed on to the real file server.

---

[2] A compliant process is one written to operate correctly within the nested process model. Noncompliant processes are undefined in their behavior and will likely result in a so-called "insanity trap" causing the process to be terminated.

2.1.4 Interposition

This use of intermediaries to support key interfaces is a fundamental aspect of the nested process model. What enables this interposition is the IPC-based nature of the interfaces, the abstract naming of objects, and the opacity of references. A child process cannot determine, if the interface is fully and correctly supported, whether it is dealing with an actual server or an intermediary.[3] The IPC system, described in greater depth in the next section, is designed specifically to support this kind of interposition. Threads may have, at any given time, one client IPC connection and one server IPC connection. In this way a thread can act both as a server to the connected client and as a client to the ultimate server.

2.2 Fluke IPC

The Fluke IPC system uses a capability model with ports as receive points and port references as send capabilities. The system supports three different messaging primitives, providing reliable connections, unreliable sends analogous to UDP datagrams, and an unusual sort of RPC having at-least-once delivery semantics for the original request message. Communication connections are between two specific threads. Each thread may have a server-side and a client-side connection.

2.2.1 Fluke Ports, Port Sets, and Port References

The principal communication object in Fluke is the **port**. Fluke ports are the receive end-points for Fluke IPC messages, roughly corresponding to receive-rights in Mach. They are analogous to having a mailbox to receive letters. Ports determine the granularity by which applications can differentiate between IPC channels. A message received on a port provides no inherent indication of the sender. Thus, servers must provide a different port for each client if they wish to distinguish between clients.

[3]In traditional security models, this sort of masked interposition could be considered a serious problem. However, Fluke's security model, Flask, leverages off this interposition and uses alternate mechanisms to provide security.

Ports are associated with a Fluke **port set**. Port sets provide a mechanism to group several ports to be serviced by a single receive operation. In fact, a server "listens to" a port set and not a specific port, though a port set may possibly have only a single port. A server may give each port an "alias" which will be returned by the receive operation to the service thread. From this alias, the server can determine from which port the incoming message was received. The port set mechanism also allows several threads to listen to the same grouped set of ports for incoming IPC messages, allowing a pool of service threads to handle incoming messages. In this way, port sets provide the actual rendezvous mechanism between the sending and receiving threads engaged in IPC.

As Figure 2.2 illustrates, messages are sent to a port via a **port reference**. A sending application holds a reference object that provides a kernel link to the specific port object. Holding a port reference is analogous to having the address of a mailbox. With that address, one can send a letter to the person who owns the mailbox. To send a message to the owner of a port, the sending application invokes an IPC send operation, passing the reference as a parameter. The kernel follows the reference's link to find the specific port. From the port structure, the kernel can find the port set in which this port is included. From the port set, the kernel can pick one of the listening server threads to which to deliver the message.

Messages are sent by the invoking thread, and are received by a thread listening to the port set that contains the port indicated by the sender's port reference. After the message is received, a logical connection is established between the sending and receiving threads, as illustrated in Figure 2.3. All subsequent messaging occurs directly between these two threads. The port, port set, and port reference, are now out of the picture.

Fluke references are opaque. They cannot be "dereferenced." That is, the holder of a reference can determine the *type* of object referenced but cannot determine the actual object that is referenced, nor the owner of the object. In this way, the port and mailbox analogy is more like a post-office box. Having the address of a post-office box provides no information about who receives mail from that

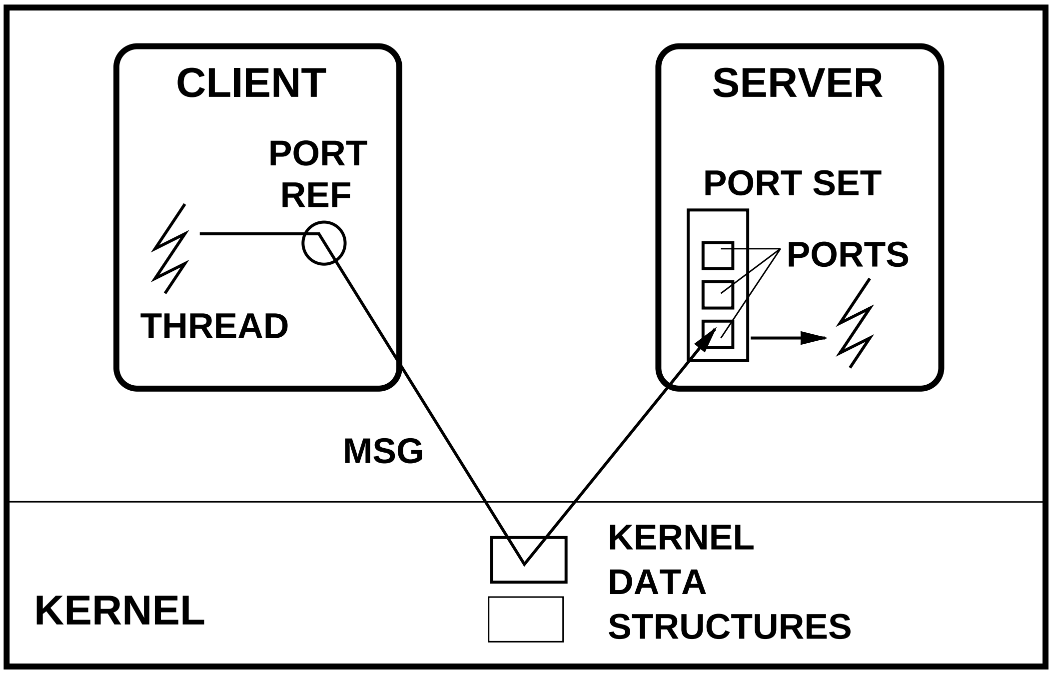

Figure 2.2. Relation between Fluke ports, port sets, and port references.

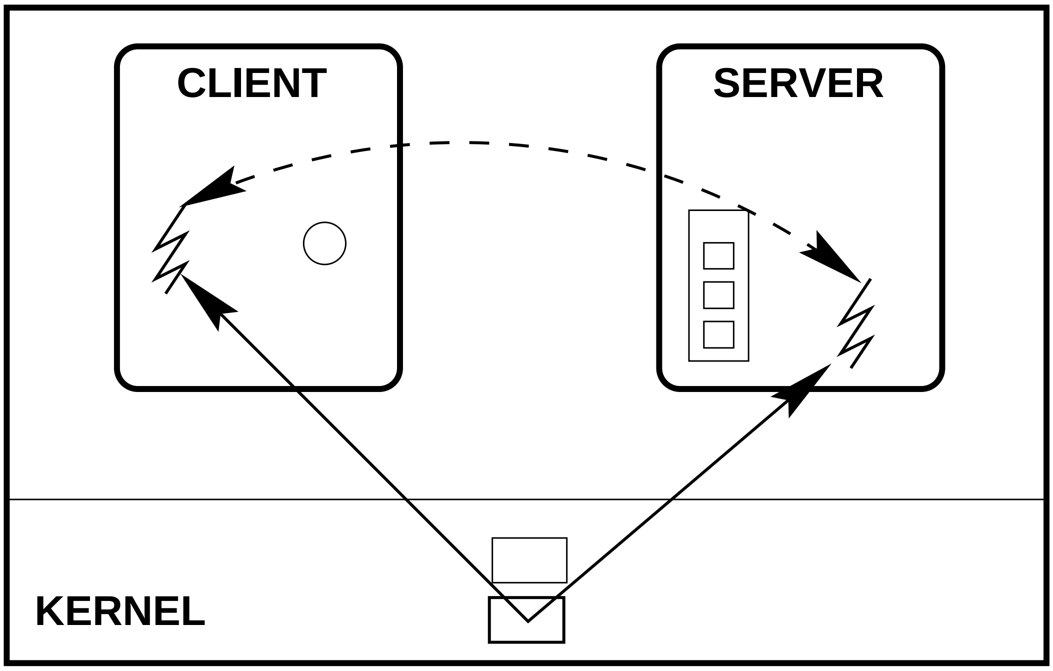

Figure 2.3. Logical connection between sending and receiving threads.

post-office box.

References are mediated by, but not controlled by, the kernel. They may be copied and transfered essentially at will. References are passed via IPC channels. It is important to note that transferring a reference is essentially a copy operation and does not invalidate the sender's reference. Since reference transfer is not controlled, once a reference to a port has been created and sent to another process, the creating process (and owner of the port) has no further control over that reference, other than to destroy the port and thereby invalidate all references to that port.

2.2.2 IPC parameters

All Fluke IPC operations take a pointer to a `fluke_ipc_params_t` structure. This structure describes the send and receive data. It contains a primary send IPC buffer, as well as an optional, arbitrary (though implementationally limited) list of additional send IPC buffers and a count of the additional buffers. A Fluke IPC buffer consists of a pointer to the application-supplied data buffer of arbitrary (though, again, implementationally limited) size, and the size in words of that buffer. Additionally, the structure contains an optional list of references to be passed with the message and the count of references to be passed. The structure also contains symmetric information for the receive operation.

2.2.3 Scatter-gather and IPC buffer management

Multiple send and receive buffers could be managed in several different ways. For instance, send and receive buffers could be matched one-to-one. Data from the first send buffer would be placed in the first receive buffer; data from the second send buffer placed in the second receive buffer, and so on. Fluke IPC specifies that send buffers are completely drained in the order presented, and that receive buffers are filled in the order presented. A receive operation returns when the send has completed or if all the provided receive buffers have been completely filled. In this latter case, the receive operation will indicate that there is more data to receive, while the sender will be blocked until the receiver provides more receive buffers. When it does so, the send continues at the point where it was interrupted.

Clients and servers are free to manage their buffers independently yet are still guaranteed to transfer all their data in order. This buffer management strategy provides a natural scatter-gather mechanism. The sender can send one large buffer, while the receiver provides numerous smaller buffers. All of the sent data will be transferred to the receiver and will be divided up into the receiver's buffers. Conversely, a sender could provide numerous small buffers that would all be transferred in order into a single large receive buffer.

2.2.4 Fluke IPC API

As mentioned above, the Fluke API provides three different "flavors" of interprocess communication. By separating the different types of IPC each can be optimized individually and applications can choose the interface with the semantics best suited for their communication needs. Fluke provides "reliable," "idempotent," and "oneway" messages. Reliable communication provides in-order, exactly-once connection-oriented semantics, optimized for client-server and RPC-style applications. "Idempotent" IPC provides a kind of RPC with at-least-once delivery semantics of the original request message and a single reply or acknowledgment. Although not itself providing idempotence, it is designed for operations that are inherently idempotent in their behavior. Finally, oneway IPC provides at-most-once delivery semantics, analogous to UDP over an unreliable network.

2.2.4.1 Reliable Communication

Fluke reliable IPC is a connection-oriented communication model with reliable, exactly-once, in-order message delivery. A reliable connection can be kept open as long as both client and server agree to do so; either the client or the server may disconnect at any time. Connections may also be broken unwittingly or in error at any time.

The reliable connection is actually a half-duplex channel, which works as follows: the client connects to a server and sends a message using one or more individual send operations. The client then indicates that it is finished sending, "reverses" the channel, and waits for the server's reply. Reversing the channel is an indication by the sender that it is finished sending. This reversal is accomplished with an IPC `over()` operation, analogous to the two-way radio protocol of signaling that one is finished talking by saying "over." Every reversal of a connection by the sender's `over()` call requires an IPC `ack()` by the current receiver before the connection can be reversed. This `ack()` is necessary to properly manage the two states that a thread involved in IPC may be in: either sending or receiving.

A server originally waits on a port set for a connection to be established on one of its ports. The server again returns from its wait after a client has connected and completed its send, or if its receive buffers have filled up. The server then processes the client's message and replies using one or more individual send operations. If the connection is to remain open, the server reverses the connection and waits for further messages from the client.

A typical RPC operation involves a client send, followed by a client over, and then a client receive. The server would return from waiting in a receive operation with the client data and would acknowledge the reversal of the connection with a server ack. When ready, the server would send the reply message. For a "traditional" RPC operation, the server would disconnect. For a long-lived connection, the server would reverse the connection and listen for the next message from the client.

An RPC operation as described would involve at least three kernel calls per side. However, the Fluke IPC API provides bundling of common-case operations, such as the client-side combined connect, send, and receive operation:

$$\texttt{fluke\_ipc\_client\_connect\_send\_over\_receive}(). \tag{2.1}$$

Note that where it is not implicitly obvious by the operation, such as with the server side `fluke_ipc_wait_receive()`, the IPC system call indicates by name the role of the invoking thread as either client or server. This specific naming is driven by the fact that each thread may have both a client-side and a server-side connection.

2.2.4.2 Idempotent Communication

Fluke idempotent IPC is designed for single RPC-type communications that require a simple reply or acknowledgment. What distinguishes this from a normal RPC operation is that the IPC may be canceled at any point and restarted, even after the original message has been received and processed by the server. It is essential, then, that repeat messages should not damage the server, i.e., change significant state, if the message is received more than once. Idempotent IPC, then, is designed for messages that have idempotent semantics. File read requests, for example, might be reasonable idempotent operations. Although Fluke idempotent IPC appears to be a kind of RPC operation, it is necessary to understand that it does not have normal RPC semantics in that the original message may be received by the server more than once.

2.2.4.3 One-Way Communication

Fluke oneway IPC provides unreliable, at-most-once delivery semantics with asynchronous sends for single messages that do not need a reply or acknowledgment. Fluke provides oneway IPC as a sort of "minimal" operation, corresponding to the capabilities of common network hardware and analogous to the UDP/IP protocol. It provides a base for applications to create their own, possibly more complex, IPC semantics if those provided by Fluke are insufficient for their needs.

2.2.5 IPC Connections

Logically, and initially, IPC is directed through a port reference to a specific port. However, ultimately the communication is between two threads. Although there is no way to specify a specific thread to which to send a message, the established connection is between two specific threads: the client thread that sent the message and established the connection, and the server thread that received the message from the port. There is no notion of the exact other thread involved in the IPC connection, yet a connection is still between two *specific* threads, as shown in Figure 2.3.

This thread-to-thread connection model is an element of persistent reliable connections. The motivation for persistent connections is that it allows repeated IPC operations between two parties to avoid having to establish a new connection for each IPC operation. Establishing a connection involves several kernel lookups to get from the port reference passed in the IPC send operation, to the referenced port, to its containing port set, and finally to a waiting thread to receive the message. By establishing the connection between the sending and receiving threads, these two threads can now maintain a persistent connection between them.

Fluke threads can maintain two reliable connections at any given time. This is the minimal number necessary to support basic client-server environments, such as an application talking to a file server that in turn talks to a disk server. A thread can have a client connection, in which the thread is a client, and a server connection, in which the thread is acting as a server. These roles are distinguished in the IPC interface by including the role name in the IPC operation system call. For instance, client operations are all prefixed with `fluke_ipc_client_`.

Attempting to receive another connection message while acting as a server, or connecting to another server while already connected to a server, will cause the existing connection to be disconnected. As an afterthought, and in deference to single-threaded applications, Fluke provides the ability for a thread to get and set its IPC client and server links. This functionality allows applications to shelve and restore IPC connections. Additionally, a thread can perform an idempotent RPC or oneway message at any time, including while it is engaged in reliable connections.

CHAPTER 3

RELATED SYSTEMS

Most modern operating systems provide some mechanism by which independent applications can communicate. Facilities are usually provided for both local and internode IPC. Operating systems that were developed from scratch to support a distributed environment, such as Amoeba and V, integrate remote IPC into their basic kernel IPC architecture. Other systems, such as Unix, provide a separate interface for remote communication. Still others have extended the local IPC mechanisms in various ways after the fact to support remote IPC, as Mach did.

All of these approaches have had to address the same basic issues, such as the naming and abstraction of communication endpoints, the location of the remote IPC module — either in or out of the kernel, and the kinds of messaging protocols provided. Certain approaches have their own particular complications, as in the difficulties of extending existing local messaging semantics to remote IPC when those semantics are difficult to guarantee in a distributed environment.

This chapter gives a brief overview of several other operating systems' local and remote interprocess communication models as reference for later discussions of the Fluke network IPC system.

3.1 UNIX

Unix chose to provide different mechanisms for intra- and internode communication. There are a variety of local communication options, including shared memory, shared files, and pipes, each with its own limitations. When faced with providing remote communication, the Berkeley Unix developers chose to create an entirely new kernel mechanism: Unix *sockets*.

Unix sockets provide a communication channel between client and server applications running on separate machines.[1] A socket connection is made to a particular endpoint. A connection endpoint is a specific Unix *port* on a specific machine. A Unix port is just an integer "address" that a server "listens" to. Certain Unix port numbers are reserved for specific services, such as `ftp` and `telnet`. Specific servers "bind," or reserve, specific port numbers as their local address. To contact a particular server, a client must specify the particular machine the server application is running on and the particular port that server is listening to.

Sockets provide both a connection-oriented byte-stream protocol such as TCP, and a connectionless datagram protocol (UDP). In the connection-oriented protocol, sockets may use the basic Unix I/O interface of `read()` and `write()` on a particular socket *descriptor*. The connectionless protocol uses more of a message-oriented interface of `sendto()` and `recvfrom()`.

3.2 Mach

Mach was originally conceived as a distributed system but was, in fact, developed entirely on a single node. Its use as the basis for a distributed environment came much later in its development. Remote communication was provided by extending the local IPC semantics, which was a central part of the Mach concept. This proved to be quite complicated. Mach's local IPC mechanism had developed into a complex system of functionality and guarantees which were reasonable to provide in a stand-alone system but quite difficult in a distributed environment.

Mach's fundamental communication object is the *port*. A port is a communication endpoint. It also encapsulates other functionality, such as message queuing. A port also provides abstraction. Unlike Unix, where the receiver is specifically

[1]The Unix socket mechanism was developed for remote communication, but it can be used for local communication as well. BSD provides a "Unix domain" socket for local communication that has similar semantics to TCP/IP (reliable byte-stream) sockets but is a purely local communication mechanism and is not interchangeable with Internet domain sockets. Internet domain sockets can be used for local communication as well, by specifying a local server's port and the local host's address as the endpoint. This is a case of "extending" network communication to local IPC.

identified, a port allows the actual receiver of messages from the port to be separated from the communication endpoint. Fluke's IPC model is quite similar to Mach's. In fact, the Fluke IPC system was developed based on extensive experience with Mach.

Each port has a set of *rights* associated with it. Port rights are capabilities associated with the port. For each port, there is one *receive right* and an arbitrary number of *send rights*. The holder of the receive right for a port can receive messages destined for that port. The holder of a send right can send messages to that port.

Mach provides connectionless messaging that requires a send right to send a message. In order to support the common client-server request-reply protocol of RPC, Mach implements a special kind of send right called a *send-once right*. The send-once right allows the holder to send exactly one message to the holder of the receive right for the port, after which the send-once right expires. To support an RPC operation, a client would send a message to a server and include in that message a send-once right, with which the server could send its reply back to the client. Later Mach implementations provided a first class RPC operation using a migrating thread model that showed a significant improvement in performance and a reduction in complexity of the IPC system [9].

All port rights, including receive rights, are transferable through IPC channels. Sending the receive right effectively migrates the port to the receiver, as all new and as-yet undelivered messages would from then on be sent to the new holder of the receive right. Mach IPC provided a rich set of options and guarantees, including port naming and renaming, guaranteed in-order message delivery, and message buffering control. Additionally, Mach guaranteed that that there would be timely notifications to the holder of the port's receive right when there were no longer any valid send rights for the port [10, 2].

Mach and its IPC system have gone through several permutations over time. The original IPC interface grew over time and was eventually replaced with a completely new interface in Mach 3.0 [4]. The Mach 3.0 IPC interface uses one system call for all messaging operations, including sending a message, receiving a message, or both. The `mach_msg()` call has seven parameters, including an options flag with thirteen options, and thirty-five return error codes!

There have also been several distinct implementations of remote IPC for Mach. This continued reimplementation is due both to the fact that the IPC interface has changed and, more importantly, because the complexity and guarantees of the Mach IPC model made all of the implementations difficult and with generally poor performance. These different systems have approached the basic problem of extending Mach local IPC in different ways, including in- versus out-of-kernel implementations, and support for a general distributed environment versus supporting only a fixed set of nodes. The following sections describe the various implementations of remote IPC for Mach.

3.2.1 Netmsgserver

The first attempt at providing remote IPC for Mach was the Mach [11], implemented on Mach 2.5, a monolithic implementation of Mach "inside" BSD Unix. The netmsgserver was a user-level server that provided all of Mach's IPC semantics for remote IPC. It was based on the notion of *network ports.* The netmsgserver maintained local ports and mapped them to network ports. The network port contained the necessary information required to forward local IPC messages as well as maintain the state information necessary to provide complete Mach IPC semantics for a remote port. The netmsgserver, in turn, used Mach 2.5's internal TCP/IP implementation for network communication [12].

The netmsgserver proved to be an extremely complicated system, having seven different major components providing network IPC messaging, simple nameservice, port right transfer management, and notification support, among others. Due to its complexity, it was never implemented for Mach 3.0 [13]. (In section 5.2.2 we roughly quantify it complexity compared to the Fluke netipc implementation.) According to [12], the netmsgserver was also very inefficient, with RPC performance measured at three to five times slower than other "comparable systems of the time." Presumably these "comparable systems" are operating systems such as Amoeba and V, which incorporated distributed IPC support as part of their base semantics and implementation.

3.2.2 NORMA IPC

The next implementation of remote IPC in Mach was provided by an in-kernel module called NORMA IPC, for No Remote Memory Access IPC [12]. NORMA IPC was developed at CMU as a PhD project. Although it correctly provided all of Mach's IPC semantics in a distributed environment, the NORMA developers chose to trade flexibility for improved performance. The NORMA system was limited to a static set of nodes that had to be known in advance. It also did not tolerate node failure or network partitions. Although this assumption is perhaps unreasonable in a more widely distributed environment, it was suited for loosely-coupled multiprocessors and clustered environments. This assumption also allowed for a marked improvement in the overall efficiency of Mach remote IPC, relative to the earlier netmsgserver. This decision to support only a static configuration improves the performance of remote IPC, but it severely limits the applicability of the system.

NORMA IPC uses global port identifiers. Global port identifiers enable the NORMA system to easily support another Mach IPC guarantee, that any two send rights will have the same identifier if they are for the same port. These identifiers also encode port location information, which allows ports to be located efficiently. If a port receive-right is moved, however, the global identifier must be changed.

NORMA's in-kernel representation uses a structure called a proxy port. The proxy port structure maintains all of the local port information, such as reference counts. It also has the global identifier for the actual port. Messages sent to proxy ports are intercepted in the kernel at the point at which they would otherwise be queued in a local port's message queue, and are instead redirected to the remote IPC system.

To solve many of the difficult issues pertaining to distributed systems, such as distributed reference counting, NORMA used a periodic token that was passed in fixed order among all of the nodes in the system. Attached to this token would be relevant information about the state of ports in the system, such as reference counts on each node. The token is passed around the system three times in a "period." On the first pass, nodes write to the token. On the second, nodes read from the token. The third pass just indicates that all nodes in the system have seen the information. Using this token mechanism, NORMA can support no-senders notifications, port death notifications, and receive-right migration.

3.2.3 Mach NetIPC

Mach NetIPC is yet another version of remote IPC for Mach [14]. Mach NetIPC was developed on top of the x-kernel [15], a network protocol implementation system. To build on top of the x-kernel system, Mach's IPC semantics were broken down into four distinct protocols: the message handler, the port manager, the send right transfer protocol, and the receive-right transfer protocol. Each of the high-level protocols is implemented in a module, and is then further broken down into subprotocols.

All messaging in the NetIPC system is mapped onto an RPC protocol. Mach messages that transfer a send-once right are easily mapped to the RPC protocol. For other types of messaging, the NetIPC system must create a mapping. For instance, single send operations are mapped to RPC by generating an empty reply message.

Although some experimentation was done with moving a minor part of the protocol implementation into the kernel in order to analyze the impact on performance, the majority of the Mach NetIPC implementation was always run in user mode, like the Mach netmsgserver and Fluke netipc.

By building on top of the x-kernel protocol system, NetIPC was able to support a general Mach-based distributed environment with reasonable performance. The designers of Mach NetIPC, however, identified areas in Mach IPC that they felt complicated the IPC implementation, primarily relating to the Mach message format and its use of typed data.

3.2.4 AD2 DIPC

AD2 is yet another version Mach targeted to support a "high-performance single system image of Unix for massively parallel processing environments." [16] To this end the implementation re-wrote much of the Mach system and added two new subsystems: the XMM memory system and the DIPC distributed IPC system. The DIPC system is essentially a complete redesign of the NORMA system, with which the OSF designers had considerable experience. DIPC is an integrated in-kernel implementation of remote Mach IPC.

The DIPC system provides three different first-class types of ports: *local*, *principal*, and *proxy*. A local port is a port for which all its send rights are located on the same machine. A local port is converted to a principal port when a send right is transferred to a remote node. A proxy is created on a node when send rights for a remote principal port are received on that node. Applications that have logical send rights to a remote principal port actually have local send rights to the proxy port. The proxy port is a kind of local port that also contains information that identifies and locates its principal port. In this way the AD2 kernel can efficiently manage local and remote ports and IPC operations.

DIPC manages the distributed reference counting issue with an object called a *transit*. A transit is a *potential* send right to a port. When a local port gets promoted to a principal port by the transfer of a send right to another node, the principal port's node also delivers a transit to the receiving node. If the receiving node should attempt to create another send right to the proxy because the holder of the first send right attempts to copy or send that right, the node must request additional transits from the principal port's node. This request will usually be filled by returning a batch of transits so that the proxy node does not have to request an additional transit for each new proxy send right. If the proxy's node should later attempt to create more send rights to the proxy than it has transits, it must again request another batch from the principal node. When the receiving node no longer has any local send right to the proxy port, it destroys the proxy and returns its pool of transits.

If a proxy's node forwards a send right to the proxy to another node, the proxy's node must send along a transit as well. The new node will create its local proxy and again manage its transits as above. When all proxies have been destroyed and all transits returned, the principal port can receive a no-senders notification.

To support receive-right transfer, DIPC added two additional *states* that a port can be in: `network` and `migrating`. A receive-right is associated with a principal port. When that receive right is transferred, the principal port is changed to the `network` state. On the receiving side, if there was an existing proxy, its state is changed to `migrating`. If there was no proxy port, a “null” port is created and set to `migrating` state. After that step is accomplished, the principal port’s state is also changed to `migrating`. At that point, the original principal port’s message queue is transfered to the new principal port. After that is complete, the new principal’s state is set back to normal, and the old principal is converted to a proxy port. These different states are required to ensure proper message ordering and transfer of the message queues. Depending on the port’s state, any senders to that port may have to be blocked until the port right transition can complete.

3.3 Amoeba

The Amoeba operating system [2] was developed originally at Vrije University in Amsterdam, The Netherlands, to study issues in parallel and distributed computing. The primary goal was a transparent distributed operating system. To that end, its IPC system was designed from scratch primarily as a remote communication mechanism. Local communication was a special case of the general IPC mechanism. The version described here is Amoeba 5.2.

Amoeba uses a capability model in an RPC framework. Unlike Mach, Amoeba’s capabilities are not kernel mediated. The capabilities are 128-bit objects that encode a server port, the particular object to which the capability refers, a rights mask, and a check field to validate the capability. The kernel maintains a mapping of server ports to particular servers on particular machines. If it does not currently have a mapping, it locates the server using a broadcast protocol. The remaining parts of the capability are used by the particular server and are ignored by the kernel. The servers themselves actually pick their own 48-bit port number.

The RPC framework is built on a request-reply messaging system, which is in turn built at the lowest level on individual send and receive primitives. Although it is possible to do so, applications do not use the lowest level primitives except in exceptional circumstances. Servers define a procedural interface for their services, and procedure stubs are used to pack and unpack the request and reply messages.

The request-reply framework has three operations, including `get_request()` and `put_reply()` on the server side, and `trans()` for the client. A server performs a `get_request()` on its port, traps into the kernel, and blocks until a client request comes in. A client executes a `trans()` to the server's port, which then blocks. The server is released when the client request message is delivered. After servicing the request, the server sends the reply with `put_reply()`.

The port number chosen by a server is actually a private port, its so-called "get port." A `get_request(port)` operation registers that port with the kernel as a get port. The kernel then computes the "put port" using a one-way function on the get port. This put port is stored in the kernel's internal tables. When a client application contacts the nameservice mechanism to locate a server, it gets back the put port computed from the server's chosen get port. All client `trans()` operations use the put port. The kernel then compares the message destination port with its internal tables to find the correct server. This mechanism was developed as a security mechanism to prevent arbitrary applications from receiving messages on the server's chosen port.

3.4 V

V [17] was developed from scratch as a distributed kernel meant to be used for a network of diskless workstations. It has a message-based communication model using a synchronous send-reply protocol. With its emphasis on the distributed environment, the IPC mechanism is a fundamental component of the system. Both local and nonlocal IPC are handled within the kernel using a single interface.

V provides a simple and efficient send and reply mechanism for short, fixed-length messages, and a supplemental bulk data-transfer mechanism. The sender calls `Send(msg, PID)` which sends the message to the specified process.[2] The message header must include the sender's PID. The sender then blocks waiting for the reply. The receiver, having issued a previous `Receive(recv-msg)`, un-blocks, processes the request, and calls `reply(msg, PID)`. The sender's original message buffer is over-written by the reply.

Bulk data transfer uses `MoveTo(destpid, dest, src, count)` which transfers `count` bytes from `src` in the sender's address space to `dest` in `destpid`'s address space. The receiver must have provided write access to the specified data range as part of the corresponding `Send()` or `Receive()` operation, and must be waiting for a reply from that prior operation. This is a "push" operation. The converse "pull" operation uses the `MoveFrom(srcpid, dest, src, count)` call, in which case, the source must have provided read access to the specified range.

Messages are sent directly to a specific process, using a system-wide unique process identifier. The process identifier encodes the "logical host identifier" which either directly encodes the host's network address, or else maps to the actual address in a host identifier table maintained in each kernel. The remaining portion of the process identifier is a locally unique identifier that identifies the particular process on that host. V was designed for a local-area network, so maintaining such unique "global" process identifiers was feasible. Nameservice is provided by `SetPid(logical_id, PID, scope)` and `GetPid(logical_id)` kernel calls. The `SetPid()` operation creates a mapping from the logical identifier, such as *fileserver*, to the actual PID. Lookups are broadcast on the local network if the local kernel does not yet have a mapping for a particular logical identifier.

It was found that the optimized short messages and bulk data transfer was inefficient for the common operation of transferring a page of data (from a file read operation, for example), V modified the original interface to include special `ReceiveWithSegment()` and `ReplyWithSegment()` operations, as well as modified `Send()` to allow the specification and transfer of an additional page-length segment.

[2]What V calls a "process" is generally referred to as a "thread" in other operating systems. V's version of what is typically known as a process is called a "task."

The use of short fixed-length packets simplifies kernel buffering. The synchronous nature of the messaging system allows a minimal protocol for reliable network messaging which used the reply as the acknowledgment of the send. The bulk data transfer also uses a minimal "blast" protocol wherein up to thirty-two network packets are transmitted and acknowledged as a group. The protocol maintains information about the last correctly-received packet so that retransmission, if necessary, begins after that last correct packet

3.5 Spring

Spring [18] is an object-oriented operating system developed at Sun Microsystems. Spring's interprocess communication is achieved via cross-address space object method invocation [19]. In this way it is architecturally different from the above message-based communication models, yet the underlying communication is fundamentally an RPC operation, essentially identical to the "migrating-threads" RPC added to later versions of Mach.

Spring's cross-address space invocation mechanism is called a *door,* analogous to Mach ports. Doors provide the necessary stack and control transfer operations to implement the cross-address space method invocation. A "door identifier" is a capability to perform a method invocation on the corresponding door. Processes, or *domains* as they known in Spring, are granted access to a specific door by the target domain, or by another domain that has valid access.

The remote invocation implementation uses a *network proxy* mechanism to support internode object invocation. A network proxy is a user-level domain that provides the same method interface as the remote server. The network proxy forwards a local door invocation to a network proxy on the server's machine, which, in turn, invokes the local door to the server.

Doors are reference counted. Whenever access is granted to a door, the door's reference count is incremented. Whenever an access is lost, either by destroying the door access, or due to domain failure or exiting, the reference count is decremented. Additionally, the reference count is incremented whenever a door is invoked to avoid a potential race condition between a door becoming unreferenced and an incoming invocation.

3.6 Summary

All of these systems have had to address the technical issues of providing network communication. Each approach has its own set of issues. In Unix, the separate implementation of remote communication was not burdened by the requirement that remote communication should be the logical extension of existing local IPC mechanisms. However, this approach forces applications to specifically identify the exact machine location of the communication endpoint. Systems built from scratch with an integrated local and remote IPC system are free to design their mechanisms to best suit their needs, but local IPC performance may possibly suffer under the burden of protocols required for network communication. After-the-fact extension of local IPC mechanisms to remote IPC may be excessively complicated due to the costs of guaranteeing local semantics in a distributed environment. In-kernel implementations potentially complicate local IPC paths, whereas out-of-kernel implementations suffer performance problems associated with having to bounce remotely-destined messages out of the kernel to user space and then back through the kernel to the network.

CHAPTER 4

THE DESIGN AND IMPLEMENTATION OF THE FLUKE NETWORK IPC SYSTEM

Fluke network IPC is provided by the Fluke Distributed IPC Manager, or DIM. The DIM is a multithreaded pseudo-server that acts as an intermediary between communicating processes that are located on separate nodes. It is not a true server in the sense that no requests are made directly to the DIM. Rather, it operates in the background to transparently provide network IPC. The DIM receives local IPC messages destined for a port or thread on a remote node and forwards them over the network to the remote node's DIM, which subsequently sends a local IPC to the ultimate receiver. In this way, processes can communicate using standard IPC mechanisms without concern for the actual location of the receiver.

This chapter describes the design and implementation of the network IPC system for the Fluke operating system. In this chapter, the issues involved in the design decisions for the network IPC system are not presented, in order to provide a more concise description of the system. Those issues are addressed in the following chapter.

4.1 Proxy Ports

In a distributed Fluke environment, there are logically two types of ports. A **principal port** is an process' actual local port. The DIM maintains **proxy ports**, which are special-purpose ports serviced by the DIM. Proxy ports are the fundamental abstraction of Fluke network IPC and are the main objects maintained by the DIM. Note that the notion of two types of ports is for discussion and differentiation purposes. This distinction is not part of the Fluke specification and is not visible to the Fluke kernel. Proxy ports are simply objects that contain a regular local port as well as additional information regarding the location of the principal port.

When the local DIM receives a transferred port reference in a local IPC message, it holds that reference and inserts a **remote reference** into the network message. The issue of how the DIM is invoked for remote communication in the first place is addressed in section 4.7. The remote reference includes addressing information for the node on which the principal port resides and an opaque key. That key is returned as the destination for incoming network IPC connections and is used by the local node to find its held reference for the principal port.[1] Note that a reference received by a DIM in a local IPC message may, in fact, be a reference to one of the DIM's proxy ports, or it may potentially be a reference to an object other than a port. These issues are discussed in detail in section 5.1.2.

When a DIM receives a remote reference in a network message, it first checks to see if it already knows about that remote port by comparing the key and addressing information to its set of existing proxies. If a proxy port has already been created for this remote reference, the DIM simply forwards a reference to the existing proxy. If not, the DIM creates a new proxy port and forwards a reference to the new proxy. The proxy port contains a local port, the remote reference key, addressing information for the principal port's home node, and administrative data.

Proxy ports are grouped together in a port set. Messages received on the proxy port set are locally-initiated client-side IPC messages. Currently, all proxy ports are treated the same, and are therefore grouped into one port set. Different classes of proxies, such as those for local-cluster remote ports versus those supporting a wider distribution requiring IP servicing, for example, could be grouped into separate port sets to simplify message handling. This differentiation of classes of proxy ports, however, has not yet been implemented.

[1]This is a possible security hole. Guessing a correct key could give an attacker access to a remote object through the netipc system. Security issues were not addressed in this work, though it is assumed that suitable controls could prevent such an attack.

4.2 System Organization

The DIM is a multithreaded service module with its threads grouped into logical subsystems. This architecture is shown in Figure 4.1. The primary group of threads consists of the IPC service threads. Additionally, there are bootstrapping and control threads and a network manager thread. The bootstrapping threads handle the announcement of a node's availability and the discovery of other Fluke-based systems. Control threads provide additional support, such as handling requests for packet retransmission on behalf of blocked IPC service threads.

The network manager thread, or netmsg_handler, receives all incoming (Fluke IPC) network messages. It must determine whether the packet is part of a reply message, in which case it must wake and hand the packet to a particular waiting thread, or if it is the first packet of a remotely-initiated message, in which case it picks one of the idle client proxy threads from the thread pool. The network manager thread does only as much processing as necessary for each packet to avoid becoming a bottleneck.

There are two pools of IPC service threads. These are the server proxy and client proxy threads that handle locally-initiated IPC and incoming network connections, respectively. The server proxy threads service the proxy port set and forward local IPC messages to the appropriate node. The client proxy threads wait for incoming network messages and then contact the local server through local IPC channels.

Every service thread has a *handle.* The handle contains all the necessary state and interfaces for a service thread to operate and be contacted. It contains the thread-type specific interface to the network manager, which includes the condition variable on which the thread waits for notification of the arrival of network messages. The handle's network interface structure also includes a "put" function that implements the message transfer operation between the netmsg_handler and the service thread. In this way, the netmsg_handler thread does not have to worry about differences between service threads, such as packet receive queue lengths and tolerance to duplicate packets. The handle also contains reliable packet protocol state, a partially prefilled acknowledgment packet, and helper thread pointers. The

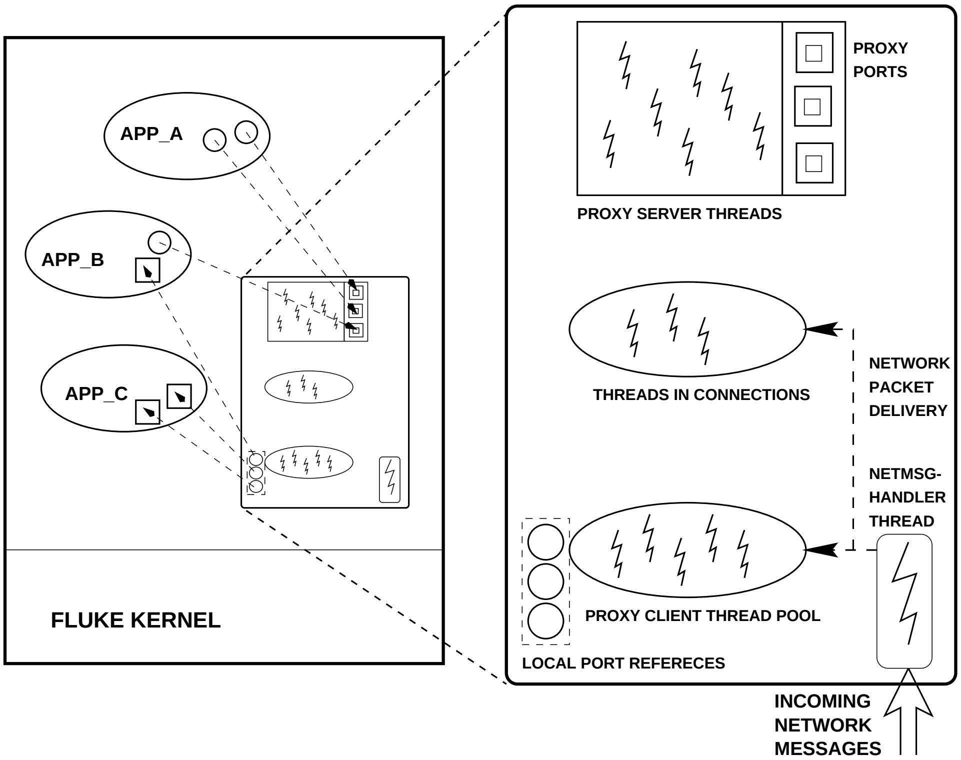

Figure 4.1. The organization of the Fluke network IPC system.

handle is used as the destination in reply messages, which allows the network manager to quickly find the correct thread to which to deliver the message. The thread's handle is passed around through most routines, and every network message has the thread's handle in the *reply_to* header field.

Figure 4.2 illustrates the steps involved in a remote IPC operation. In step 1, local client processes send client-side IPC messages for remote ports to the reference they hold. These messages are sent via the Fluke kernel in step 2 to the proxy port created for that remote principal port. One of the local server proxy threads will be woken by the receipt of the IPC message. The port alias returned to the woken thread from the `wait_receive` call indicates on which proxy port this message was received. The alias is a pointer to the proxy structure that contains the relevant information about the remote port, as mentioned above. In step 3, the message is then forwarded over the network to the appropriate node. The network message is then delivered via the netmsg_handler thread to a client proxy thread in step 4.

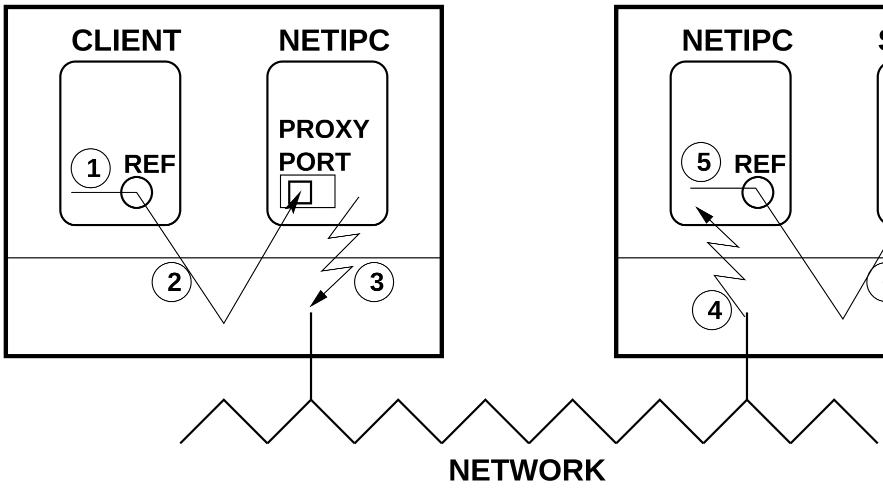

Figure 4.2. Sending a remote IPC message.

That thread finds the local server port and sends a local IPC message to the server in steps 5 and 6.

Figure 4.3 shows the logical connections between threads that are established after the initial IPC send operation. The client thread is connected to a local netipc system server proxy thread. On the remote node, the netipc client proxy thread is connected to the the server thread. There is also an implicit connection between the server proxy and client proxy threads. The server's reply is sent back along this chain of thread connections to the client.

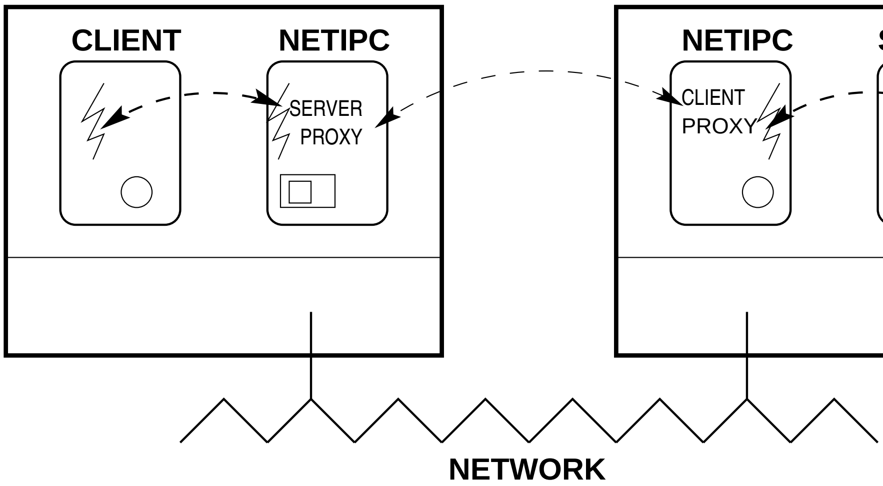

Figure 4.3. Sending a remote IPC message.

For oneway IPC's, the receiving server proxy thread simply fills in and sends the network message, and then returns to service more locally-generated IPC messages from the proxy port set. For idempotent and reliable messages, the local server proxy thread must wait for an ack and possibly a reply message. In these cases, after sending the first network message, the thread blocks and waits for the reply message. The thread sets the "reply-to" field in the network packet headers so the reply message can be directed to the appropriate local thread.

For reliable and idempotent IPC's, the connection with the sending client thread must be maintained by the receiving server proxy thread for the duration of the reliable IPC connection or idempotent RPC. This arrangement causes a DIM thread to be consumed by each connection that is established, which can potentially lead to resource depletion and many idle, blocked threads if connections are maintained during long periods of inactivity. The thread pools create additional service threads when the number of available threads drops below a specified threshold.

The DIM network message header includes a destination field. The destination of a network packet is either a local server port or a specific thread that is already involved in the IPC connection to which this particular packet belongs. If this packet is part of an initial remote client connection, the destination field contains a remote port reference key. The packet is passed to a client proxy thread, which uses the remote port reference key in the destination field to look up the actual principal port. The client proxy thread then connects to the local server. This chosen thread must maintain the connection with the local server as long as the connection remains open.

4.3 Code Path

For reliable IPC, both the server proxy and client proxy threads travel essentially the same high-level code path, only starting at opposite ends. The path for an IPC connection is a loop, starting and ending either at the network interface or at the local IPC interface. This loop is shown in Figure 4.4.

In the service of all flavors of Fluke IPC, threads travel some portion of the

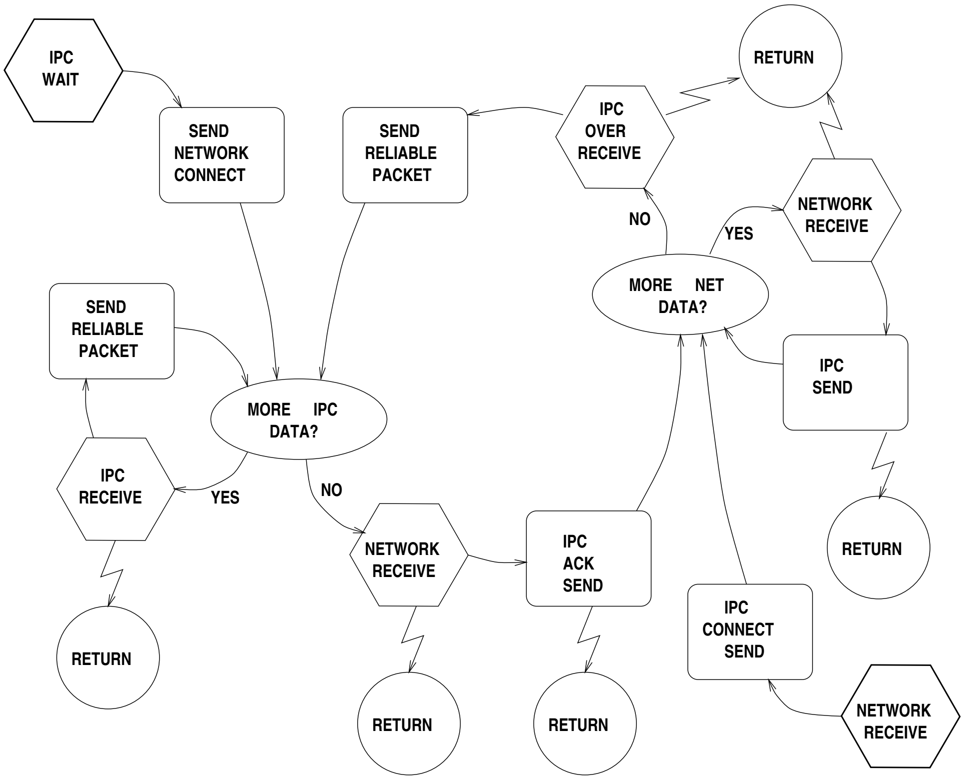

Figure 4.4. Distributed IPC code path

loop, with server proxy threads starting at the local IPC interface, and client proxy threads starting at the network interface. Threads servicing oneway messages only travel one side, and idempotent messages only make one turn. There are minor differences in the actual code a thread invokes depending on whether the thread is a server proxy or client proxy service thread, and on the flavor of IPC. All of the circles labeled `RETURN` in the diagram are points where a disconnection may occur. In the diagram, hexagonal boxes are blocking points and rectangular boxes are send operations, either an IPC send or sending a network packet.

Following a locally-initiated reliable IPC message, the code path is as follows. One of the server proxy threads waiting on the local proxy port set returns from its `wait_receive` operation, labeled `IPC WAIT` in the upper left corner of the diagram. This thread finds the remote destination information via the proxy port's alias and fills out the network packet's headers. Received data have been marshalled into the network packet by the IPC operation. If there is room, any received references will have their corresponding remote references marshalled into the network packet after the IPC data. The thread then sends the network packet. If there are more data or references to receive, the thread enters an inner loop to receive the IPC data and send the network packets, shown in the smaller loop on the center-left of the diagram. After it has received all the local data, the thread blocks and waits for the remote reply. Processing the network reply is similar to the handling of incoming network messages, described next.

When a network message is received, the network manager thread determines if the message is a reply message for a specific waiting thread, or if it is a new connection, in which case it is handed to one of the pool of idle client proxy threads, blocked waiting for a network message in the lower right corner of the diagram. Once the receiving thread is picked, the packet is placed in that thread's receive queue and the thread is signaled. The receiving thread wakes and first handles any received remote references by creating proxies, adding them to the proxy port set, and referencing them. These references are added to the local IPC message and the whole message is sent via the appropriate IPC flavor and method. Once again, the unmarshalling of data into the client process's IPC buffers is handled by the local IPC system. The thread repeats the network receive operation until the connection is reversed or disconnected. If the connection remains open, the thread blocks in an IPC receive operation and then works its way along the local IPC receive path as described above.

4.4 Passing References to Nonport Objects

The Fluke reference object can reference most other Fluke objects, and references are passed via IPC channels. Therefore, the Fluke IPC system allows threads to pass references to arbitrary Fluke objects. Currently it does not make sense to pass references to objects other than ports to a remote node, as those references would be meaningless without additional mechanisms to interpret them. For instance, a reference to a memory region on one node would make have no meaning to the memory manager on the receiving node. Additionally, if the reference were passed as a remote reference as described above, the receiving node would create a useless proxy port. To handle this, the sending side thread must check all passed references to verify that they are references to ports. References to objects other than ports are currently simply ignored. This issue is discussed in detail in section 5.1.2.

4.5 Reference Propagation

An process may forward a reference it has received through IPC channels from another process. In a distributed environment, this requires some additional consideration. For instance, suppose an process on machine A sends a reference to an process on machine B, which causes the creation of a proxy port on node B. Then the process on node B forwards its reference, which is in fact a reference to the proxy, on to another process on node C. This reference transfer causes the creation of a proxy on node C. Naively, node C's proxy would reference the proxy on node B. A message from C would eventually get to A, but it would first go through node B. Additionally, the proxy on node B could be destroyed, which would leave node C's proxy orphaned. Node C's proxy should instead be a proxy for the principal port on node A, not for the proxy on node B.

The DIM on machine B must recognize the out-going reference as referencing one of its proxies. Instead of sending a reference to the proxy, the DIM must send a remote reference for the principal. The proxy structure contains all the necessary information to generate such a remote reference. The sending DIM, therefore, will always examine the passed reference for type, as above, and also for the specific object type to determine if it is a reference to one of its proxies. With these two checks on the sending side, the receiving DIM only checks incoming remote references to see if it has already created a proxy for that reference.

4.6 Network Interface

Fluke provides a simple read and write interface to the network. This is provided by the C library's open, read, and write calls, which support user-level access to the network through the kernel. Writes are synchronous and involve an IPC to the kernel. Reads are blocking, and also involve an IPC operation. The DIM opens a specific named network device and simply reads and and writes network packets. The lowest levels of the netipc system use this read/write interface.

The DIM network packet, or *netmsg*, contains a network-specific header, an IP header, the DIM header, and a reliable packet protocol header. These headers are followed by as much client data and as many remote references as will fit in the packet after the headers. The maximum size of a netmsg is equal to the network's Maximum Transmission Unit.

The DIM's high-level network interface is designed such that the lower layers might be easily replaced or modified. A custom reliable packet protocol was implemented to support this work. That protocol is mostly hidden from the service threads, though there are occasions where exposing protocol information to the service threads and IPC state to the protocol solved particular trouble spots. For instance, the first packet sent by a server proxy for a reliable connection must be through `send_connect_packet()`, whereas all following packets are sent through `send_reliable_netmsg()`, regardless of whether they are logically part of the initial send operation or a part of a reply message.

All threads waiting for a reply message call `get_netmsg()` which performs protocol processing and returns the appropriate next netmsg. All client proxy threads also must use `get_next_netmsg()` as their entry point into the code loop. This is different from `get_netmsg()` used when a thread is waiting for a reply on an existing connection. These situation-specific routines allow the protocol to clear and reset the thread's per-connection state.

4.7 Bootstrapping Communication

All of the discussion prior to this point has assumed that client processes have had references to server ports in order to initiate IPC communication. Without an initial reference, there is no way for any communication to take place. However, as discussed earlier, references are transferred via IPC. So how does a client get its first reference?

All processes receive an initial reference to their parent. Using this reference processes query their parents and build their environment. In Mach, processes also received a reference to a name server, from which they could request references for named services. The Fluke system does not provide such a name server. Instead, the file system is used to provide a namespace. Servers mount a port in the file system. A lookup in the file system for the named server returns a reference to the port mounted by that server. In this way, clients are able to retrieve a reference to the server.

The netipc system leverages off this naming mechanism. When the DIM starts, it acquires a reference to the local file system. As part of its initialization, the DIM broadcasts its existence over the network. In that announcement message, the DIM exports its reference to the local node's file system. Remote nodes that are already up and running receive this message and mount a proxy for the exported file system under the machine's name in the specially-named directory `/fdi`. The remote node acknowledges the announcement message and sends its reference to its own file system with the acknowledgment. The initializing DIM gathers the acknowledgment messages and mounts a proxy for each node in its local file system under its own `/fdi` directory.

This bootstrapping and announcement mechanism builds the distributed namespace. A client looking up a server on a remote node first looks up the remote node's machine name in the `/fdi` directory. That lookup returns a reference to the proxy that represents the file system on the remote node. The remote server will have mounted an access port in the remote file system. Once the client has a reference to the remote file system, the lookup proceeds just as if the client were looking up a server locally, except that each lookup passes through the netipc system. The lookup may involve several RPC's to the remote file system as each component of the path is traversed. Each RPC returns a reference which causes the creation of a proxy on the local node. Ultimately, the lookup will return the remote server's reference, and the client can begin communication with the remote server through the netipc system.

CHAPTER 5

EVALUATION OF FLUKE'S SUPPORT FOR REMOTE IPC

The Fluke IPC architecture is based on a capability model. Although this architecture is neither new nor conceptually different from several other message-oriented operating systems' IPC mechanisms, its semantics and certain implementation details are different from these other systems. These variations led to different solutions for remote IPC in each of the operating systems. The architecture and semantics of the Fluke IPC system were expected to enable a simple and straightforward implementation of transparent remote IPC. For some aspects this proved to be true. For others, there were added challenges that complicated the implementation of completely transparent remote IPC.

This chapter evaluates the architecture, mechanisms, and semantics of Fluke and its IPC system with respect to their extension over a network for remote communication in a distributed Fluke environment. Individual elements of Fluke IPC are evaluated in terms of their cost or benefit to the remote IPC implementation. Where appropriate, other operating systems' remote IPC mechanisms are compared and contrasted with the Fluke mechanisms and network IPC system to illustrate the architectural and semantic differences and the issues that arise from those variations.

5.1 Capabilities

Fluke's capability model for its IPC system is based on the abstraction of *ports* and *port references.* A port is the receive point for IPC messages, and a port reference is the destination to which to send a message. Fluke's use of such a model as the underlying architecture for its IPC systems is not unique. Other operating systems have had similar architectures for their IPC systems. The capability model was chosen for the Fluke operating system because the attributes of this model are well-suited to many of the architectural goals of the Fluke system, including nested processes. The main quality of capabilities that is desirable for the Fluke architecture is that they are opaque. An opaque capability provides no information about the underlying object. For IPC, this means that the holder of a port reference has no direct information about that port, including its location or owner. Equally, a port provides no direct information about the sender of messages received on that port.

The importance of this opacity to Fluke is that it allows intermediaries to *interpose* on IPC channels. Because communication via port and port reference does not give any information about the message sender and receiver, the ultimate communicating entities in an IPC channel are unaware of possible intermediaries. This is a fundamental aspect of Fluke's nested process model, which relies on interposition on IPC channels to monitor and control nested processes.

The absence of naming and location information in ports and port references prevents a process from knowing just where it stands in a nesting "hierarchy." A process does not know, from its references to server ports, whether it runs, for instance, directly under a process manager with all rights and privileges thereto, or under one or more interposing controllers or monitors.

These same qualities of capabilities allow, in theory, the IPC model to be transparently extended across a network to support a distributed environment. A reference to a particular port gives no indication of the machine location of that port. With transparent remote IPC, applications can be written that are based solely on IPC and are independent of their location or the location of their servers or clients. In the same way that two communicating applications cannot directly determine if there is an intermediary interposing on their IPC channel, two applications cannot directly determine if they are located on the same or separate machines.

A demonstrated example of this is file access. An application written to perform file operations ran unchanged for both local disk and remote disk file operations simply by replacing its file-system reference from the local file system to the file system on a remote machine.

Transparent interposition allows an out-of-kernel, proxy-based implementation of remote IPC. This frees the kernel's local IPC implementation from the burden of differentiating and managing both local and remote IPC. As a rough comparison, consider line counts for IPC-related files in the kernel. Fluke's local IPC system requires 10 kernel files with a normalized total line count of 3367 lines. In Mach 3.0, on the other hand, not including its in-kernel NORMA files, there are 46 IPC-related files containing a normalized line count of 11,852 lines.

5.1.1 Capability Transfer and Port Migration

The transfer of a port reference to another process bootstraps Fluke communication. Once a process has a reference for a port, it can send messages to that port and, consequently, to the server listening to that port. Passing port references equivalently bootstraps the network IPC system. A reference received in a local IPC message is marshalled into the network message as a remote reference. On the remote node, a received remote reference drives proxy creation.

Mach has a notion equivalent to the port reference in its port *rights*. A Mach port has several different kind of rights associated with it: the receive right, which allows the holder to receive messages from that port; send rights, which allow the holder to send messages to the port; and send-once rights, which are only valid for one send operation. The send-once right allows a single reply-message to support a remote procedure call abstraction. All of the Mach port rights can be passed through IPC messages, including the receive right. Transferring the receive right for a port essentially, although not literally, migrates that port to another process.

Fluke ports, on the other hand, do not migrate; i.e., they cannot be transferred to another process. They may be moved to a different location within a process's address space using `fluke_port_move()`, but there is no explicit mechanism by which they can be transfered to another process. Port migration in Fluke could be handled fairly simply in the local case by adjusting kernel data structures. In the distributed case, it would be considerably more complex. Several issues would be involved if ports were allowed to be transferred between processes.

First, if there were any references to the port on the original home node, a proxy would have to be created and those references would have to be adjusted to reference the proxy instead of the original port. Given a mechanism to transfer ports, this would not be overly difficult. The original port would simply be held by the DIM, inserting it into the proxy structure. The local references would then not require adjusting and would automatically reference the proxy.

Subsequently, the message that included the port transfer would be sent to the new home node for the port. The DIM would recognize the arrival of a port, and generate its own reference to the port before passing in on. This is the reverse of passing a reference to a local port out to a remote node, in which the passed reference is held by the DIM, and a proxy is created on the remote side.

Once a port has moved, nodes having proxies for that port need to be made aware that the port has moved. Nodes could be alerted of port migrations by broadcast, specific notification of all nodes known to have a proxy reference to the port, or by lazy notification of nodes when they actually try to send a message to the port at its original node.

There are of course tradeoffs between these approaches. Both broadcast and specific notification could generate a burst of network traffic with notification messages and acknowledgments. Broadcasting also announces to the world that the port has moved, possibly violating privacy concerns. Additionally, specific notification may be inadequate, as a remote node may have forwarded the reference, in which case the list of nodes to which to send notifications would be out of date. Lazy notification would force the original home node to keep bookkeeping data around until it was sure all nodes had learned of the port's migration. This is not so much of an issue if there are outstanding references to the port on the original home node, in which case that node will have to maintain the proxy structure. However, if there are no local references, the original home node would want to know when it is safe to destroy that information. This is a reference counting or garbage collection issue, discussed further in section 5.1.3.

Because Fluke IPC connections are between two threads, moving a port does not necessarily affect threads that are already engaged in connections that were established on that port. Those connections could likely complete without incident.

Mach, on the other hand, guarantees the in-order delivery of all messages even if the port moves. Mach must go to considerable effort to move the message queues with the port in a two-phase operation. In the first phase, the port is marked as migrating, and any new messages are queued as before, but are not delivered. The receiving node would continue to send messages to the sending node until it received the receive right which is marked as migrating. At that point, message sends are blocked until the right has been completely transferred. Also during the first phase, all other remote nodes are notified of the migration of the receive right. These nodes would send messages to the original home node until they receive notification that the right has moved. During the second phase, the contents of the original message queue is pulled over by the receiving node. After that is complete the system is stable, and local message sends on the receive side are unblocked [16].

Since the transfer of the port cannot be completely atomic with respect to all the nodes in the system, new connect messages would have to be handled appropriately. A simple solution would be to reject the connection and include the new node information in the NACK. However, that information may not be available yet. Depending on the notification mechanism, that connect would have to be blocked either at the original home node until it received the new node's remote key for the port, or at the node invoking the IPC until it received the port migration notification.

Although there may be some value in allowing ports to be transferred, the added complexity to both local-case and especially remote IPC led the Fluke designers to leave it out of the Fluke specification, thereby significantly reducing the difficulty of providing remote Fluke IPC.

5.1.2 Fluke Reference Objects

In most of this discussion, the *reference* capability discussed is a *port reference.* However, reference objects in Fluke are, in fact, first-class objects and can "reference" almost all of the other Fluke object types. The netipc system, however, is only concerned with ports and port references. The complication is that references of all types, not just to ports, can be passed through IPC channels.

In Mach *all* objects are represented by ports, and all operations on those objects involve an IPC on the object's port to the server managing the given object. Although servicing a Fluke object may still involve IPC to a server, the reference to the object to be serviced is passed to the server, rather than making a distinct IPC operation on the object's port as in Mach.

Passing references through IPC channels is a natural mechanism to pass object accessibility from one application to another. In same-node IPC, passing a reference to an arbitrary object is not an issue, as references are kernel objects and can be easily resolved to the actual object. However, for remote IPC, passing a reference to an object other than a port does not necessarily make any sense.

In Mach, passing the send right to, for example, a memory object over the network causes the creation of a proxy port for that memory object. The operation of the network IPC system is not affected by the kind of object for which it is generating the proxy. The passed right simply causes the creation of a proxy. IPC operations on that proxy will proceed normally. However, if the memory server on the sending node does not support distributed memory, the requested operation on the proxy will fail, and the proxy port for the memory object is useless on the remote node.

In Fluke, a netipc-passed reference to a memory region would, naively, cause the creation of a proxy port on the remote node. However, the region reference clearly does not correspond to a proxy port. Additionally, operations on Fluke memory objects use the memory reference rather than invoking IPC operations on the memory object's port, as in Mach.

This led to some issues about how to treat the passing of references to objects other than ports. The first naive solution was to do nothing and simply pass the references on, causing the creation of a bogus proxy port on the remote node. This was attractive because it required no intervention on the part of the netipc system and did not change the IPC semantics. But it is clearly wasteful, as these proxy ports would never be used, and was rejected.

A second approach was to simply reject IPC messages that transferred nonport references. This would clearly violate the “transparent distribution” expectation of remote IPC, giving applications the knowledge that they are dealing with a remote service. It does make sense that an application may want to know this, so that it does not repeatedly try to send a nonport reference to a remote application that would be unable to use the reference. One difficulty with this solution is that there is currently no agreed protocol to return useful information to the application that this was a failure because of the type of transferred reference. All the netipc system could do is disconnect, which would be returned as a disconnect error, possibly prompting the client application to repeatedly try to send the reference anyway.

A third solution would be to pass along tag information about the kind of object referenced, such that it may be processed on the remote node in some manner that makes sense, such as supporting some kind of call-out mechanism. This is possibly the desired solution, but such “advanced processing” does not exist. (See chapter 6 for a discussion of this mechanism.) Therefore, the chosen solution was to simply ignore nonport references for the time being. Although this may not be the best solution in the long run, it does not preclude more advanced reference handling mechanisms and does not suffer from most of the above limitations.

This solution, however, does limit the transparency of remote communication in a similar manner to outright rejection of the message. By simply ignoring the reference and not passing it to remote nodes, the semantics of remote IPC are no longer the same as for local IPC. It becomes possible for applications to determine, by evaluating the behavior of passing references to objects other than ports, that they are communicating with a remote process.

As in Mach, other systems that use capabilities for IPC have a limited and specific role for those capabilities. They are only to access the object or server via IPC. In Amoeba, a capability specified a particular object in a particular server. Applications would use that capability to identify the object to which they were sending a message. Passing a capability is straightforward, as the capability continues to only provide access to the object to the holder of the capability. Spring doors are much the same. They identify and give access to a particular object of a particular server.

Fluke's general-purpose reference object adds additional checking for each passed reference to the remote message transfer path. It also makes the implementation of transparent remote IPC for the complete set of local IPC operations, and in particular passing nonport references, next to impossible for the netipc system to provide by itself. To do so, the system would have to understand the semantics of distribution for all Fluke objects and be a distributed object manager for all of them. This is clearly both impractical and undesirable. The alternative is to have a mechanism by which the netipc system acts in conjunction with distributed object managers to handle the passing of references to object other than ports to remote nodes. Ultimately, general-purpose references are the single greatest impediment to the netipc system's ability to provide completely transparent remote IPC.

5.1.3 Reference Counting and Notifications

Many operating systems provide reference counting in their object model [18, 16, 4]. This facility allows applications to determine when there are no more possible senders for a given receive point, which allows the application to discontinue listening to that receive point and reclaim resources. This is a valuable service. In the local case, it is straightforward to provide when all references are kernel-mediated objects. It can be considerably more complex in the distributed case [12, 20].

Mechanisms for distributed reference counting are generally built on top of reliable local reference counting. Systems that guarantee accurate reference counts, such as Mach, must go to considerable effort to provide a correct count of all the references in the system. Often there is a tradeoff between flexibility and efficiency, such as Barrera's NORMA IPC, which supported only a fixed set of nodes and did not tolerate failures and network partitions [12]. Given the difficulty of the problem, other systems do not attempt to provide an accurate count. Instead, these systems only determine if there are any *possible* outstanding references [16], allowing eventual, rather than prompt, notification to the application of no more senders. The complicating issue is that references to an object may be passed around or copied between remote nodes without the knowledge or control of the node on which the referenced object resides.

Although local kernel-supported reference counting is straightforward, it does incur some kernel overhead. The kernel must perform object lookups, validation, and access for each passed reference. Additionally, whereas reference counting is a valuable service to applications that need it, not all applications will want kernel-provided reference counts or want to pay the price for the additional kernel overhead that would be required for reference counting. Such applications may either not need reference counts, provide it for themselves, or rely on a higher-level protocol to provide it as necessary.

The designers of Fluke had considerable experience with Mach and its IPC system. They believed that much of the complexity of that system, as well as the great complexity of the various network IPC implementations for Mach, was driven by the requirement to provide accurate reference counts and timely notifications of no-more-senders. They also believed that many applications would not want reference counting, and that adequate reference counting for those applications that did could be provided by an outside service. Certainly there are algorithms for distributed reference counting available, but, for the stated reasons, reference counting was not supported by the Fluke kernel. Instead, a higher level protocol, called "MOM" (Mini Object Model) [21], would be available for use by those applications that needed reference counting. The netipc system, which was designed after the original designs for Fluke were complete, both benefited and suffered from this decision.

Not guaranteeing reference counting frees the kernel and the netipc system from the responsibility of having to keep an accurate count of all references throughout the distributed system. This would appear to be a significant relief. However, the lack of local reference counting in fact poses several problems for the netipc system. Primarily, the netipc system *itself* needs reference counting. It needs local reference counting for proxy ports so that it can release the proxy when the local reference count drops to zero. It also needs distributed reference counting so that it can determine when it no longer needs to hold a passed-in reference to a remote port. Whereas the netipc system needs this for its own resource management, this also brings up an additional issue. The netipc system would have to participate in any higher-level reference counting mechanism.

Fluke applications use a run-time system implementing MOM on top of Fluke to support reference counting. Without going into great detail about this system, it supports object reference counting and no-senders notifications using additional threads. This system has proven to be quite complex and error-prone.

There was considerable debate about the mechanism by which this runtime system would provide notifications. All of the proposals would have bearing on the netipc system, as it would have to, at minimum, cooperate with the runtime to provide notifications in the distributed environment. At worst, the netipc system would have to be re-engineered to accommodate protocols required by the runtime system. At the time the netipc system was implemented, the notification portion of the runtime system had not been completed, and therefore the netipc system neither provides, nor has access to, a reference counting and notification system.[1]

5.2 IPC Interface

The IPC system supports three different messaging semantics, two concurrent IPC connections per thread, and persistent half-duplex connections, all through an exceptionally broad interface. The interface is broad in that there are distinct kernel calls for each IPC operation. The specific call depends on the operation type, the messaging semantics, the particular role in the IPC connection as either a client or server, and the state of the IPC connection. Although there are Fluke

[1]Currently, the basic Fluke notification and runtime system is complete but not fully robust.

kernel semantic requirements and implementation issues that make such a broad interface desirable, as discussed at length in [22], this broad IPC interface is rather unwieldy for a low-level user such as the netipc system. However, much of this interface is hidden from the application-level programmer who writes to an RPC interface and uses the Flick IDL compiler [23].

5.2.1 Fluke IPC flavors

The Fluke IPC interface separates the three different "flavors" of messaging semantics provided by the IPC architecture. Providing different messaging semantics allows applications to choose the messaging semantics best suited to their particular needs. It also solved some of the sticky problems associated with other aspects of the Fluke architecture, such as the requirement that threads handle their own memory faults [8]. The intent of the separate interfaces was to allow efficient implementations for each that did not rely on runtime parameter evaluation to determine the particular semantics requested for each message operation.

To provide transparent remote IPC, the network IPC system must support each of the three interfaces and provide the same messaging semantics for each for remote messages. Assuming a suitable network protocol, this in itself is not a great challenge and in fact provides the same opportunity to optimize each path. But there are aspects of the different interfaces for the different flavors that make supporting the semantics of each flavor as they are specified in the design documentation somewhat complicated and, in certain cases, incomplete.

5.2.1.1 One-way

Fluke's oneway IPC mechanism provides at-most-once delivery semantics, corresponding to a simple network packet send operation. It provides no guarantee of delivery or of the return of an error code. That is, messages may be silently dropped without generating an error. If an error code is returned, the client application may assume that the error is valid, but a successful return does not necessarily indicate successful message delivery. There are also no guarantees about message size. According to the design specification, a oneway IPC message with more data than will fit in a network packet may be either silently dropped or truncated. The IPC design specifies that a client may determine the MTU allowed for both oneway and idempotent IPC sends using the idempotent IPC interface. This issue will be discussed in the following section.

As one might expect, oneway IPC is essentially trivial to extend to the remote case. The network IPC server simply receives the message on the proxy port and sends a network packet to the remote node, where it is forwarded by the remote node's network IPC server to the appropriate port via oneway IPC. If the original message is larger than what will fit in a network packet, the local network IPC server can either ignore the message altogether and return to wait for another IPC invocation, or silently truncate the message and send only what will fit in a network packet.

The specification of the return codes from a Fluke IPC `wait_receive` operation, however, does not provide enough information to the receiving thread. It cannot determine that the invoking IPC operation was a oneway send, and that the sender attempted to send more data than the receiver's receive buffers could hold. Given this wrinkle, oneway sends that are larger than the network MTU are never dropped by the network IPC server but are always silently truncated and sent. This, of course, does not guarantee the delivery of even the truncated message, as the network packet may be lost.

5.2.1.2 Idempotent

As mentioned earlier, Fluke idempotent IPC has slightly odd semantics. The motivation for these semantics is that it allows the kernel to cancel and restart from scratch the IPC operation at any time up to the successful completion of the reply to the client. The justification for this behavior is that it allows the kernel to use idempotent IPC operations to handle error conditions, such as a page fault.

This cancellable and restartable behavior may suggest some difficulty for implementing remote IPC, but there is, in fact, no undue complication for the netipc server. The important factor is the possible timings of the cancel and restart operations, and of when the IPC calls can return. Once the netipc server has received a local idempotent IPC call, it reliably sends the single network packet to the remote node. The remote netipc server initiates an idempotent IPC operation, from which it does not return until the operation has successfully completed or there was an unrecoverable error.

Although the remote idempotent operation may be canceled and restarted, once the remote client proxy returns, the operation has successfully completed. The remote netipc server then reliably forwards the reply to the local server, which completes the RPC with a reply and returns to service the proxy ports. Although that local reply invariably returns successfully, there is no guarantee that the invoking client actually received the reply. In the event that the IPC was canceled in the reply phase, due to a page fault in the client, for example, the IPC would simply be restarted. The netipc server would see a fresh incoming idempotent IPC, which would be handled the same as if it were the original request. Attempting to deliver the reply to the first request would fail, and the reply is simply dropped.

As mentioned above, the Fluke specification states that applications can use idempotent IPC to determine the network MTU. However, the return code from a `wait_receive` operation only indicates that this was an idempotent IPC operation. It does not indicate that the sender attempted to send more data than was received by the receiver, as can by indicated in a reliable IPC return code. Whereas the Fluke specification states that the MTU can be determined through idempotent IPC, it does not specify how that information is returned. Additionally, given that there is no indication in the idempotent return code that there are more data, the netipc system has no way of knowing that the sender tried to send too much data, so it does not know to return some indication of the MTU. Additionally, there is no specification of the protocol for returning that value to the sender.

The Fluke specification does not specify the maximum message size for idempotent and oneway operations, other than to specify that it is guaranteed to be the same for all idempotent and oneway messages. Given the described limitation of the IPC return codes, the proposed solution is to arbitrarily reduce the netipc MTU for idempotent and oneway messages. This is reasonable, since the netipc

MTU is somewhat arbitrary in the first place, not necessarily directly corresponding the the actual network MTU because of required netipc packet headers, which may change. The MTU would be defined to be just less than the actual available payload space in a network packet. The netipc server can then determine from the count of remaining space in its receive buffer, whether or not the invoking idempotent (or oneway) operation was attempting to send more than the maximum allowed message size. However, since a protocol for servers to return error conditions was not defined, this aspect of the network IPC server was not implemented. Instead, idempotent (and oneway) messages are limited to the amount of data that will fit in a single packet. Messages are silently truncated if they are too long.

5.2.1.3 Reliable

Fluke reliable IPC provides exactly-once, in-order message delivery of arbitrary messages, as well as a sustainable, half-duplex connection between two specific communicating threads. Due to its semantics, reliable IPC is the most challenging and interesting IPC flavor with respect to providing remote IPC. The elements of Fluke reliable IPC will be discussed in turn.

5.2.1.3.1 In-order Delivery

A fundamental tenet of an RPC system, or any reliable communication mechanism, is the reliable and ordered delivery of messages. On a single node, this is not difficult to provide. In the remote case, over an unreliable network, it is more problematic. Certainly, a reliable network would alleviate much of the difficulty; another option would be to use a standard reliable network protocol, such as TCP/IP. This choice would certainly be a viable development alternative, but in a local area network cluster, this design is likely too heavyweight for an efficient netipc implementation.

To provide reliable distributed IPC over an unreliable network, some reliable network packet protocol is necessary. Minimally, the protocol must support recovery of lost packets and the ability to order the packets such that the higher levels of the netipc system receive all of the packets of the message in order for delivery to the application through local reliable IPC channels. Given such a reliable packet protocol, providing reliable remote Fluke IPC is reduced to providing the rest of Fluke's reliable IPC semantics in a distributed environment.

5.2.1.3.2 Thread-to-thread Connections

Fluke reliable IPC provides a reliable half-duplex connection between two *specific* communicating threads. The significance for distributed IPC is that the proxy thread that received the initial IPC message must be the one that completes the connection. For a client proxy involved in a simple remote RPC connection from off the network, this is not an issue, as the thread will execute the Fluke reliable IPC to the server and then send the reply over the network to the originating node. When it does become an issue is for the server proxy threads and for both proxy thread types during long-running reliable connections. The specific difficulty is the need to deliver all reply and long-running connection network packets to the specific thread responsible for servicing that particular remote IPC connection.

There are several approaches to solving this issue. The most desirable approach would be an efficient packet-filtering mechanism whereby threads could register with the network implementation to receive all packets destined for them. Such a mechanism would have to be lightweight and support efficient registering and unregistering of thread-specific filters. A packet filtering mechanism for the Fluke network implementation has been designed but not yet implemented.

Another approach would be to allow any thread to service any connection. Such an approach seemingly violates the semantics of Fluke reliable IPC, which requires connections to be between two specific threads. However, if threads could essentially masquerade as each other, such an approach would be feasible. It would require that threads be able to alter their IPC state. This functionality was not a part of the original Fluke specification, but was added after it was realized that it was valuable for single-threaded servers. With this support, server proxy threads would receive reliable IPC connection requests and forward the message across the network. After completing the initial phase of the connection, the server proxy thread would store its IPC state and connection information, and then return to service more locally-initiated IPCs. There would then be a pool of threads listening to the network. Any thread could accept an incoming packet, determine the connection, set its IPC state, and execute the appropriate reliable IPC call. After completing that phase of the IPC, the network service thread would either disconnect the reliable connection or store the IPC state, and then return to service more incoming network packets.

This approach suffers from several drawbacks. First of all, it requires as many as five additional system calls *per packet* to get, set, and save the threads' IPC state, plus saving and looking up that state. To be fair, only one of the system calls and the lookup of the saved state is on the critical path. The other calls would be made after completing a network send. This approach would also require each thread to be able to handle the different classes of network messages, including IPC messages, acknowledgments, protocol, and control messages.

Additionally, it complicates the implementation of, and interaction with, the simple reliable packet protocol. More importantly, it introduces significant synchronization requirements between the threads involved in servicing the connection. A thread receiving a packet must look up the connection information, determine if this packet is the next to be delivered, and ensure that the previous packet has been successfully delivered and the IPC state saved in a stable state, before it can set its IPC state and continue the packet delivery. If this packet arrived out of order for some reason, the thread would have to save it (presumably) and trigger some sort of protocol handling, either requests for lost packets or delivery of previously saved packets.

Ultimately, the chosen solution involved having one network message handler thread receive all incoming network packets and distribute them to the appropriate threads. Although this is a potential bottleneck, it does alleviate several of the problems inherent in other solutions. It can also easily be scaled to have several point threads listening to the network without adding additional complexity and only minimal (and likely uncontested) synchronization. Because of its critical role, the netmsg_handler thread does only minimal verification and protocol processing. This processing includes verifying the destination as either a valid local port reference or a proxy thread waiting for messages suppressing duplicate packets to prevent unnecessary thread scheduling and context switching. Additional verification and protocol handling is done by the receiving thread in layers of the protocol and network-receive code.

5.2.1.3.3 Persistent Half-Duplex Connections

Perhaps the most distinguishing feature of Fluke reliable IPC is its notion of long-running, half-duplex connections. In Fluke's half-duplex model, communication can occur in both directions, but only in one direction at a time. One thread sends until it is done, indicating that it is finished and will now be receiving data. The receiving thread receives data until it gets the sending thread's indication that it is finished, at which point it may begin to send data. This connection model fits naturally with the common-case simple RPC. It also supports a common case in which a client makes several requests to the same server without intervening requests to different servers. In such a case, the client can keep the connection open, which eliminates the need for port and thread lookups on each invocation.

The half-duplex nature of the connection minimizes the required state and complexity of the both the kernel IPC mechanisms and the network IPC system. Providing a full-duplex channel would have added much more complexity for a feature that would likely be rarely used.

The support of long-running connections, however, added some complexity to the netipc system. Persistent connections add more states and transitions to the basic message-handling code. The basic IPC interface is also made more complicated by persistent connections. Threads must manage sending, "reversing" the connection, receiving, acknowledging that the connection has been reversed, and the possibility of disconnection at any point. This complexity is partially illustrated by the code loop figure in Chapter 4, Figure 4.4. In practice on Fluke today, the use of long-running connections is uncommon. To be fair, persistent connections are not currently practical to use simply because the IDL compiler and runtime do not support them. However, it appears that the added complexity to support persistent connections, at all levels of the system, outweighs the possible benefits of eliminating a few lookup operations.

5.2.2 Complexity

The Fluke IPC interface provides kernel calls for each IPC operation. The particular call depends on the IPC flavor, the state of the IPC connection, and whether the caller is a client or server in this connection. Although this leads to a rather broad and somewhat unwieldy interface, the complexity is largely hidden from the application-level programmer by IDL and runtime support. The complexity is also mitigated by the provision of kernel calls that combine common-case IPC operations, such as reversing a connection and then listening for messages, into single calls. This separation of the interface into many explicit "micro-operations" is necessary to support interruptible and restartable kernel operations [22, 24], another important aspect of the Fluke operating system.

Whereas the IPC interface is somewhat complex and unwieldy, Fluke's overall IPC system is relatively simple, at least compared to Mach. The absence of features found in Mach-based systems, such as port migration, reference counting and notifications, allowed for less complex kernel and remote IPC systems. Although, as discussed previously, the ultimate value of this complexity versus functionality tradeoff is unclear, the Fluke kernel and remote IPC systems are clearly less complex than those of Mach-based operating systems.

To illustrate this relative complexity, normalized line counts of similar IPC systems were compared. Although line counts of code is by no means a definitive measure of system complexity, it is illustrative and provides circumstantial evidence. For this comparison, all files relating to IPC were run through a normalizing script that removes white space, comments, C preprocessor commands, lines that contain only braces, and null statement lines, such as lines with only a semicolon. The 10 IPC-related Fluke kernel files total 3,367 lines. The netipc system contains 48 files totaling 4,429 lines.[2]

[2]For reference, a simple line count yields 6,433 lines for the kernel files and 9,924 lines for the netipc system.

Mach 2.5 using the netmsgserver system consisted of 4,257 normalized lines of kernel code and 14,016 lines of netmsgserver code. Mach 3.0, using NORMA, has 11,852 lines of kernel code, and 8,034 lines of code in the NORMA system. The AD2 system, with its integrated remote IPC system, consists of 26,013 lines of IPC code. The Fluke kernel IPC system is roughly the same size as the Mach 2.5 IPC system, before the "revised" Mach 3.0 IPC interface. The Fluke kernel IPC implementation is roughly 28% the size of the Mach 3.0 kernel IPC system. Including remote IPC systems, Fluke IPC is 43% the size of Mach 2.5 IPC, 39% the size of Mach 3.0 IPC, and 30% the size of AD2's IPC system.

5.2.3 IPC Buffer Management

Fluke IPC calls accept lists of send and receive buffers. Between communicating threads, these buffers may be mismatched, in that the sizes of send buffers do not directly match the sizes of the receivers provided receive buffers. Different IPC systems have handled this problem in different ways. Some, for example, attempt to transfer using buffer pairs, copying only as much data as will fit in the receiver's buffer, and then moving on to the next pair of buffers [8]. Fluke transfers data without regard for buffer pairing. Data are copied out of the sender's buffers and into the receiver's buffers in the order presented, filling each before moving on to the next. This makes the kernel IPC implementation cleaner and more suited to cancel and restart as necessary.

It also makes aspects of the implementation of network IPC quite clean and elegant. The netipc system presents as its receive buffers the payload portion of a network packets. Large IPC messages will be copied into the network packets. On the remote node, individual network packets will be copied into the receiver's buffers. In this way, fragmentation of large messages comes essentially for free, and is completely transparent to the application. If a client and server choose to have coordinated buffer sizes, data that may have been fragmented along the way will be "regrouped" into the buffers presented by the receiver. This is clearly a desirable IPC feature for distributed IPC systems.

5.3 Performance Results

This section presents a breakdown of the performance of remote IPC on Fluke. Times are presented for roundtrip message transfer for varying message sizes for each flavor of IPC. In comparison, local IPC times are also presented. Local and remote file operations are also presented. In these prototype implementations of both the Fluke operating system and remote IPC, performance was not the primary goal and, in fact, the performance numbers are not exceptional. A brief analysis of the numbers will be presented at the end of the section. The tests were run on 200 MHz Pentium Pro-based machines over 100Mb ethernet.

5.3.1 Round-trip times

Table 5.1 presents times for round-trip message transfers. The remote operations are between a user-level client and a "ping" server. The server receives the complete message and immediately replies in the manner appropriate to the type of IPC. The message size is given in words of four bytes each. Fluke IPC has a minimum message size of two words, so no smaller messages could be timed. Times are given in microseconds. For oneway messages, a reference to a client port was previously transfered to the server to allow the server to reply using a oneway send operation.

As discussed in section 5.2.1, oneway and idempotent are limited to a single packet. The data payload portion of an netipc ethernet packet is 364 words. Any additional data are simply ignored. This leads to the leveling off of the times for idempotent and oneway IPC after 256 words.

These numbers are representative of times for round-trip remote RPC. The numbers vary significantly between repeated tests. Round-trip times and the times for individual sections of the code each vary typically by as much as 20%.

5.3.2 Code-path analysis

Table 5.2 lists a detailed breakdown of the costs of various portions of the netmsg system's reliable code path. This table presents the local perspective, and Table 5.3 presents a breakdown of the remote node netmsg system's processing of the same message. The local perspective has a single time for the "other side." This single

Table 5.1. Round-trip times in μ-secs for request-reply message transfer.

Message Size	Remote			Local		
	Reliable	Idempotent	Oneway	Reliable	Idempotent	Oneway
2	1413	1135	1185	27	23	34
4	1488	1142	1247	23	20	26
8	1461	1125	1229	22	19	25
16	1463	1129	1246	22	19	26
32	1442	1130	1248	23	19	26
64	1456	1139	1239	24	19	26
128	1612	1213	1421	24	21	27
256	1886	1414	1705	37	32	40
512	3602	1637	1971	61	55	67
1024	4533	1695	1888	141	139	124
2048	6464	1658	1879	211	204	211

number includes all the time between the local node network send of the packet to when the netmsg system receives the reply and returns from its network wait.

These times are for a 64-word message that includes a passed reference to indicate the costs involved in reference transfer and proxy creation. The tables are organized with absolute time in the first column, the time per operation in the second, a key word with which to label operations, and a brief description of the operation.

Transferring a reference adds roughly 250 microseconds to the cost of a reliable RPC. The most significant costs are in processing the locally-received reference, including checking the reference type, whether it is a reference to a proxy, and adding the reference to a hash table; marshalling the remote reference into the network packet; unmarshalling the remote reference and checking that it is a valid reference to a known node; and creating the proxy port and structure. Additionally, there are added kernel IPC costs in transferring the reference from the client to the DIM and from the remote DIM to the remote server.

Certain numbers merit discussion. The first significant cost is that of the actual network send operation, AFTER_L_SEND in Table 5.2. The user-level network interface is highly unoptimized. There are six copies along the path from the C

Table 5.2. Round-trip times in μ-secs for request-reply message transfer.

Total Time	Delta Time	Key	Description
0	-	LOCAL_START	Client issues connect_send_over_receive
31	31	SPX_RECV	Proxy server thread receives the message
103	82	TO_L_NET_SEND	Process reference and lookup remote port
210	107	AFTER_L_SEND	`net_send()` operation
1112	902	OTHER_SIDE	The "other side:" net send to net recv
1115	3	LOCAL_NMH	net_msg_handler lookup receiver
1130	15	L_NODE_LOOKUP	Lookup replying node
1136	6	THD_LOOKUP	Lookup or create replying thread info
1138	2	DUP/ADD_THD	Duplicate packet check or add thread info
1153	15	FIFO_ADD	Add to receive FIFO
1157	4	SIGNAL	Fluke condition signal
1220	63	SIG_RECV	Proxy server receives signal
1228	8	KILL_REX	Stop retransmission thread
1239	11	RESET_PROT	Reset reliable protocol
1599	360	SEND_ACK	Send disconnect acknowledgment
1618	19	CLEAR_QS	Clear any packets left in queues
1722	104	CLIENT_RECV	Client returns from IPC operation

library through the generated common protocol stubs and the COM interface to the kernel server and on to the device.

There are also significant costs involved in the the reliable packet protocol. The most egregious of these is the generation and sending of the acknowledgment of the remote server's disconnect. Other protocol costs include the handling of sequence and message id numbers, maintenance of the sent and receive queues, controlling the retransmission thread, and managing duplicate and lost packets, as well as other costs. The DUP/ADD_THD cost is significantly higher during warm up, or whenever an unrecognized remote proxy thread sends a message.

On the remote side, there is a similar path, but with some different costs. When a local node receives a reply, the netmsg_handler thread is able to determine directly to which local proxy thread the packet is destined from the reply-to information in the message header. On the remote node, at the stage labeled REM_NMH in Table

Table 5.3. Round-trip times in μ-secs for request-reply message transfer.

Total Time	Delta Time	Key	Description
0	-	REMOTE_RECV	Remote netmsg_handler receives packet
17	17	REM_NMH	netmsg_handler picks proxy client thread
26	9	R_NODE_LOOKUP	Lookup sending node
32	6	THD_LOOKUP	Lookup or create sending thread info
32	0	DUP/ADD_THD	Duplicate packet check or add thread info
49	17	FIFO_ADD	Add to receive FIFO
54	5	SIGNAL	Fluke condition signal
91	37	SIG_RECV	Proxy thread receives signal
93	2	CPX_REL	Dispatch to reliable code path
114	21	CPX_PROC	Unpacking netmsg
190	76	PXY_CREATE	Create proxy port
280	90	REM_IPC	proxy client connect_send_over_receive
299	19	TO_REM_SEND	Up to network send

5.3, the incoming packet is for a new connection, for which the netmsg_handler must pick a new client proxy thread from the thread resource pool. Note also that the processing for the check for a duplicate packet is minimal for a new connection.

5.3.3 File access

As a simple proof-of-concept example, remote file access was implemented using remote IPC, transparently accessing a remote node's filesystem, with both local and remote fileservers unchanged. The example program was able to access local and remote files in exactly the same manner, simply by changing the name of the file from a local file to a file on a remote node. This was a substantial demonstration of the transparency achieved by the netipc system.

Each netipc system exports its machine's local disk to other Fluke netipc systems. Remote file access involves simply opening, reading, and writing a file under `/fdi/`<*machine*>`/`. Table 5.4 compares local versus remote file operations. Currently, files are implemented in a memory-based file system loaded with the boot image and therefore do not include any physical disk accesses.

Table 5.4. Times in μ-secs for file access operations.

operation	remote	local
`open()`	7400	1140
`read()`	1550	95
`write()`	1540	69

The remote `read` and `write` times include the base cost of a reliable RPC to the remote node, plus the cost of the remote file operation. These times are for 32 byte read and writes. A read or write involves a reliable RPC to the kernel through IDL-generated stubs for the C library interface to the file system common protocols. Once a file has been opened, the reference exists to which the IPC messages are sent.

The `open` operation traverses the file path, which involves an IPC operation for each element in the file's path. For the local access, there is only one lookup operation to get the reference for subsequent file operations. For remote operations, there are minimally three lookups for `fdi`, <*machine*>, and finally for the remote file.

5.3.4 Performance Analysis

These timings indicate that there is considerable opportunity for performance improvement. Both the netipc system and the Fluke kernel are unoptimized prototypes. Fluke operations tend to be relatively expensive. The Fluke network interface could also use significant improvement. There are several context switches and copies along the path to the device. The many operations that require synchronization and locking, such as queue and hash-table manipulation, are also expensive. The reliable packet protocol implemented as part of this work, though conceived as a simple and efficient custom protocol, proved to be quite cumbersome and inefficient.

5.4 Evaluation Summary

In general, Fluke's use of capabilities for its communication abstraction works well for the provision of remote IPC. The general capability model extends gracefully in a distributed environment. With the abstraction of capabilities and Fluke's IPC interposition, the basic design of the network IPC system was quite elegant and simple. Fluke's IPC interface is a bit cumbersome, and it is made even more so by supporting persistent connections. The thread-to-thread nature of the connections added complexity and a performance penalty from additional context switches. This could be alleviated by an improved network interface in Fluke. The decision not to support port migration as part of the capability model significantly reduced the potential complexity of the netipc system. The decision to leave reference counting out of the kernel was a double-edged sword. Although it did relieve the network IPC system of the complexity and burden of managing distributed reference counts, it made it impossible for the netipc system to responsibly manage its own objects. Lastly, the specific nature of Fluke's capability model that uses general-purpose reference objects made complete transparency for all IPC operaions virtually impossible for the network IPC system to provide alone.

CHAPTER 6

OPEN ISSUES

Several of the issues involved in extending Fluke local IPC that have been discussed in this work are not yet resolved. This chapter will discuss some of those issues and present some possible extensions of the netipc system and changes to Fluke to benefit remote communication and distributed computing in a Fluke environment.

6.1 Reference counting

Kernel-supported reference counting was not included in the Fluke architecture primarily because it was believed that guarantees of kernel-provided reference counting and notifications would unnecessarily complicate remote IPC due to the complex nature of providing distributed reference counting. Additionally, reference counting would add kernel overhead to all applications for the benefit of only those that chose to use it. Furthermore, it was believed that adequate reference counting could be provided outside of the kernel for applications that needed that service.

Distributed reference counting is, indeed, a difficult problem. There has been considerable research in this area, primarily in the area of distributed garbage collection, which is essentially the same problem [25]. Distributed reference counting, however, is possible. Solutions range along a line of tradeoff between flexibility and generality of the system versus simplicity of the implementation. Barrera's NORMA IPC, for example, provided distributed reference counting for a static set of participating nodes and did not support node failure or network partitions [12]. The *x-kernel*-based NetIPC system used a set of protocols for reference transfer to accurately count all remote references in a dynamic system [14].

The difficulty with Fluke's decision to not include kernel-supported reference counting is that the kernel is uniquely the best location to support reference counting. Experience within the Fluke development group has shown that external mechanisms providing reference counting are complex and prone to error or incomplete implementations. External reference-counting systems also add additional overhead to systems that use their services. This is a serious performance issue to critical systems such as the netipc module that must deal with large numbers of transient objects.

The situation in Mach that caused potentially spurious no-senders notification messages was not because the kernel provided reference counting, but rather because of the nature of Mach's IPC system (see 5.1.3). Fluke would not suffer from this condition, as it uses connection-based semantics in its reliable IPC. With connections, the kernel can temporarily increment a port's reference count during a connection to avoid this condition. Spring uses this mechanism in its object model to avoid this problem [18].

Although kernel-supported local reference counting in Fluke would add some small amount of additional overhead to kernel operations, this would not be a significant amount. Kernel support would greatly benefit applications that need efficient reference counting, such as the netipc system. Certainly having the netipc system provide distributed reference counting would add some complication to the netipc system, but it is possible, provided that there is reasonable and efficient local reference counting available. With local reference counting, there are plenty of algorithms that provide distributed reference counting. A potential downside is that if reference counting is defined to be part of Fluke IPC semantics, then applications using remote IPC will always pay reference-counting's sometimes high cost in the distributed case, even if they do not need it. However, many applications, and the netipc system in particular, would greatly benefit from kernel-supported reference counting, and the Fluke kernel should provide that service.

6.2 Location Transparency

Transparent remote communication in Fluke is compromised by the limitation that references to objects other than ports are not currently transfered by the netipc system. This is a subtle but fundamental limitation of Fluke's general-purpose reference objects. Location transparence is more obviously compromised by the mechanism by which applications discover remote processes. Currently, an NFS-like interface is used, wherein servers mount themselves in their local file system, and a machine's local file system is exported by machine name by the netipc system. The Fluke file-system "mount" operation is a special operation that installs a given port at the specified location in the file system. Accessing a mount-point returns a reference to the mounted port. Clients look up servers in `/fdi/`<*machine_name*>`/`<*server_name*>. This removes any expectation of location transparency at the very start.

It would be desirable if servers could be found in a location-independent manner. One possible solution would be for each node to have a special local directory in which servers would mount themselves. For discussion, this directory can be called `/servers`, but the name is not important. The netipc system would implement `/servers` as a special directory object. Mount operations in that directory would register the server with the netipc system. As a system came on-line, the netipc system could, if desired, broadcast its list of local servers as part of its announcement message. This would generate proxies for servers on remote nodes. Client application lookups of servers would either return the local server, or the mounted proxy for that server. If a local netipc system did not have a particular server mounted in its `/servers` directory, a global lookup protocol could try to locate the server.

This would require a considerable amount of additional work for the netipc system. Essentially, the netipc system would have to provide a simple nameservice. Part of this effort would be the implementation of the special directory object. This would require the netipc system to support the entire directory interface of the Common Protocols file-system API.

Another solution would be to have nameservice provided by an external name server. This solution would be more in keeping with the microkernel organization of modularity and independent servers. This external service would still have to operate through the netipc system in order to establish the necessary proxies. One drawback to this organization would be that an independent server would not be able to mount proxy objects directly. Either the nameservice would have to work in concert with the netipc system, which would partially defeat the goal of having an independent server; the lookup mechanism, i.e., using the file system namespace, would have to change, which is contrary to the goals of the Fluke architecture; or the mount mechanism would have to change to allow the mounting of a reference as opposed to a port. Without such changes, the netipc system is best suited to supporting the distributed name service.

6.3 References

Perhaps the most interesting area of further work lies in addressing the current limitations on completely transparent remote IPC caused by the handling of Fluke general-purpose reference objects. As mentioned earlier, the Fluke general-purpose reference is the most significant obstacle to providing completely transparent remote communication. To overcome this limitation, there must be mechanisms to support the transfer and handling of arbitrary (nonport) references.

As described in section 5.1.2, the reason the netipc system does not pass references to nonport objects is that there are no mechanisms to support such references on a remote node. There is no fundamental limitation in the netipc system that prevents it from passing these references. The netipc system must already evaluate each local reference it receives for the type of referenced object, so it could easily pass the reference along with a tag indicating its type. The wire format of "remote references" is specific to the support of remote ports, but it, too, is not fundamentally limited to port references. The receiving node could evaluate each incoming remote reference for its referenced object type and only create proxy ports for references to remote ports.

The difficulty lies in how to handle remote references that are for nonport objects. Some object types make more sense than others in a distributed environment. A reference to a memory segment, for instance, might be used in support of distributed memory. But what of a reference to a thread or a memory region or a port set? Just what a reference means on a remote node is a subject of research unto itself. Given that the semantics of remote references can be defined, the question remains as to how the netipc system should support arbitrary remote references?

One possible solution would be a call-out mechanism. External servers, such as a distributed memory manager, would implement the distribution of specific kinds of objects. These servers would be registered with the netipc system, likely using the file system mount mechanism as do standard servers. When a nonport remote reference was received in an IPC message, the netipc system would perform call-out operations to the server responsible for providing the necessary support services for that type of object. These call-out operations might likely have to occur on both sending and receiving nodes, adding a definite performance penalty. Additionally, the external servers would likely have to communicate by remote IPC, leading to some synchronization as well as further performance issues.

The solution described here is just one possible mechanism to support arbitrary remote references. Other mechanisms are possible, and the definition of the semantics of remote references may motivate still other mechanisms. Given Fluke's general-purpose reference objects, completely transparent remote IPC requires the definition of some mechanism to support the passing of arbitrary references to remote nodes.

CHAPTER 7

CONCLUSION

The design and semantics of an operating system's local communication mechanisms have a direct impact on the mechanisms used for remote communication. The impact of these decisions is particularly apparent when remote communication is intended to be the transparent extension of local communication. In this case, the design choices for local IPC must be evaluated with respect not only to local performance, but also for their extensibility to a distributed environment as well.

The Fluke microkernel operating system is designed to support a novel architecture in which child processes are "nested" in their parent, which is called a "nester." All of the Fluke subsystems were designed and tailored to support this nested process architecture. An additional goal of the Fluke system is that it should support transparent distribution of computation. Fluke's IPC system is a fundamental component supporting these architectural goals. The design decisions regarding IPC were made primarily with regard to its performance in its crucial role within the system. This thesis evaluated the design of Fluke IPC and the Fluke architecture with respect to their support for remote communication, and showed that, although it has many aspects that support remote communication, the Fluke system has certain features that are difficult for remote communication and other features that make it impossible for a network IPC system to provide completely transparent remote IPC.

The basic elements of Fluke IPC are its capability model and its messaging semantics. The use of a capability model facilitates several aspects of the Fluke architecture. It also enables transparent remote communication. However, the nature of the Fluke capability model leads to some difficult issues.

For the most part, the capability model is well-suited to the Fluke architectural goals. One of the key elements of this architecture is that objects are abstractly named. For arbitrary nested processes and nesting hierarchies to operate correctly, there can be no absolute objects or namespaces. This abstraction is achieved in Fluke by its capabilities: ports and references. Capabilities contain no application-visible state that provides information about the particular object or of the owner of the object.

Another crucial element in support of Fluke's nested process architecture is transparent interposition on IPC. The capability model with its opaque references, and the design of the Fluke IPC system are important aspects of interposition. The Fluke IPC system allows each thread to have two concurrent IPC connections: one to a client and one to a server. In this way, an application can act as an intermediary on a communication channel between a client and server. By interposing on their children's communication, nesters can control the child's behavior and resources. Interposition is a fundamental aspect of the nested process architecture.

These decisions made in support of the nested process architecture are complementary to transparent remote communication. Given the microkernel organizational model of having the kernel support only the minimum essential services, higher-level services are provided through out-of-kernel servers. This dictates that remote communication, not defined to be in the minimal set of essential services, should be implemented as a user-level service. As a user-level module, the network IPC system acts as a transparent intermediary, essentially interposing on IPC channels to processes on remote nodes. The abstraction of capabilities and the support of interposition allows a straightforward and elegant transparent extension of local IPC. For the same reasons the capability model works well in support of nested processes, the model also works well for remote IPC.

The definition of transparent remote communication is that a process cannot determine from the communication mechanism or semantics whether or not the process with which it is communicating is located on the same machine. Half of this transparency is achieved by having communication mechanisms that are no different when the communicating processes are remote as they are when the processes are located within the same machine. The capability model and the Fluke IPC interface and mechanisms effectively support this aspect of transparent remote communication. Proxy ports, transparent interposition, and opaque references allow the netipc system to extend the same IPC mechanisms used in local IPC to remote communication.

The other half of transparent remote communication is that the semantics of communicating with a remote process should be the same as if the process was located on the same node. To this end, aspects of the Fluke architecture, such as its general-purpose reference objects, make completely transparent remote communication much more difficult. In fact, completely transparent remote communication is impossible to provide through the services of a network IPC system alone.

Fluke's capability model extends beyond IPC. References are first-class objects and can reference most other kernel objects. References to all types of objects can be passed through IPC channels. Because of this reference model, the netipc system must be object-aware. Since the netipc system has no means by which to process references to objects other than ports, it does not pass these references to remote nodes. In this way, the semantics of local communication do not transparently extend to remote communication. Additional mechanisms, protocols, and services would be required to fully support this model and provide completely transparent remote IPC semantics.

The designers of Fluke, having extensive experience with Mach, felt that kernel-supported reference counting added more complexity than value to the local and, especially, remote IPC systems and therefore chose to leave that service out of the kernel. Programming with capabilities, however, often leads to the use of many transient objects. This effect is especially pronounced in the netipc system, which is constantly creating proxy ports for references passed to remote nodes. Without reference counting, or some form of garbage collection, there is no way to know when it is safe to destroy an object. It is unclear whether the added complexity of providing distributed reference counts is offset by the benefits of those counts, or if such a service can reasonably be provided outside of the kernel. Systems whose performance is critical, however, such as the netipc system, must have some efficient reference counting or garbage collection in order to properly manage their resources.

Reliable IPC in Fluke is connection-oriented. IPC connections are established between the sending client thread and the receiving server thread. These connections were incorporated into Fluke's IPC design as a mechanism to support communication optimizations, wherein clients that expected to make several requests to the same server could maintain an open connection and thereby eliminate the reference $\rightarrow$ port $\rightarrow$ port set $\rightarrow$ thread lookups to establish connections for each request. This optimization could have further benefits when the communicating threads are on separate machines. The chain of lookups would have to be performed on both nodes: first to find the local proxy port and then to find the server port. There are also additional lookups in the network IPC system. The argument for persistent connections is that maintaining a connection eliminates all of these additional lookups after the first connection is established.

This potential optimization comes at a cost. With thread-to-thread connections, the thread that handled the initial request connection message must be the one to complete the reply, and potentially remain connected for further messages on this connection. The current network interface does not allow for arbitrary packet filtering, which would support early demultiplexing of netipc messages to the appropriate thread. An additional thread is required to listen to the network and do packet demultiplexing to hand off incoming network packets to the appropriate thread so that the thread-to-thread connections can be maintained. Therefore, even a "simple" request-reply RPC requires an additional context switch.

The other aspect of persistent IPC connections is the complexity added for their management. Even though the expected typical behavior of applications is to use simple, RPC-like request-reply message sequences, the netipc system must be prepared to handle persistent connections. Persistent connections complicated the the netipc system's messaging handling by adding more possible states, transitions, and error conditions, to the basic code path. It is possible that Fluke's IDL compiler and runtime might someday be extended to transparently use persistent connections for simple RPC's (by caching connections), but this is not currently planned.

Certain aspects of Fluke IPC clearly support remote IPC. The buffer management and scatter-gather semantics of the IPC interface are ideal for remote communication. The in-order processing of send and receive buffers provides "automatic" data marshalling as well as fragmentation and reassembly for large messages. The decision to not support port migration is another choice that benefited the network IPC system. Port migration would have added considerable complexity to the system.

The design and semantics of the Fluke IPC system support of remote communication well. Certain elements of remote IPC would benefit from fairly minor architecture revisions, such as the provision of in-kernel reference counting and eliminating persistent connections. The capability model extends naturally to support remote communication and a distributed environment. However, Fluke's general-purpose reference objects make completely transparent remote communication impossible for the netipc system to provide on its own. With some revision, the Fluke IPC system design and semantics would support a simple and elegant implementation of remote IPC, but the Fluke reference model makes the completely transparent extension of local IPC a challenging problem.

REFERENCES

[1] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson, "Microkernels meet recursive virtual machines," in *The Second Symposium on Operating System Design and Implementation Proceedings*, pp. 137–152, October 1996.

[2] A. Tannenbaum, *Distributed Operating Systems.* Prentice Hall, 1995.

[3] D. R. Cheriton, "The V distributed system," *Communications of the ACM*, pp. 314–333, March 1988.

[4] R. P. Draves, "A revised IPC interface," in *Proc. of the USENIX Mach Workshop*, pp. 101–121, October 1990.

[5] H. M. Levy, *Capability Based Computer Systems.* Digital Press, 1984.

[6] R. P. Goldberg, "Architecture of virtual machines," in *AFIPS Conf. Proc.*, June 1973.

[7] B. Ford and M. Hibler, "Fluke: Flexible μ-kernel environment – application programming interface reference (draft)." 110 pp. University of Utah. Postscript and HTML available under http://www.cs.utah.edu/projects/flux/-fluke/html/, September 1996.

[8] B. Ford and M. Hibler, "Fluke: Flexible μ-kernel environment – design principles and rationale." Internal design documentation, 1996.

[9] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *Proc. of the Winter 1994 USENIX Conf.*, pp. 97–114, January 1994.

[10] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso, *Programming under Mach.* Addison Wesley, 1993.

[11] D. P. Julin and M. N. Group, "Network server design." Unpublished report, Carnegie Mellon University, available at ftp://ftp.cs.cmu.edu/project/mach/-doc/unpublished/netmsgserver.ps, August 1989.

[12] J. Barrera, "A fast Mach network IPC implementation," in *Proceedings of the Mach USENIX Symposium*, pp. 1–12, 1991.

[13] D. Orr, 1993. personal communication.

[14] H. Orman, E. I. Menze, S. O'Malley, and L. Peterson, "A fast and general im-

plementation of Mach IPC in a network," in *Mach III Symposium Proceedings*, April 1993.

[15] N. Hutchinson and L. Peterson, "The x-kernel: An architecture for implementing protocols," *IEEE Transactions on Software Engineering*, vol. SE-17, pp. 64–76, Jan. 1991.

[16] B. Bryant, "Design of AD2, a distributed UNIX operating system," Tech. Rep. 1.0, Open Software Foundation Research Institute, 1995.

[17] D. 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*, pp. 129–140, October 1983.

[18] J. Mitchell, J. Gibbons, G. Hamilton, P. Kessler, Y. Khalidi, P. Kougiouris, P. Madany, M. Nelson, M. Powell, and S. Radia, "An overview of the spring system," in *Compcon Spring 1994*, February 1994.

[19] G. Hamilton and P. Kougiouris, "The Spring nucleus: A microkernel for objects," in *USENIX 1993 Summer Conference*, June 1993.

[20] R. Jones and R. Lins, *Garbage Collection.* Willey, 1996.

[21] R. McGrath, "MOM, the Mini Object Model: Specification (Draft)." July 1998. Unpublished report, available at http://www.cs.utah.edu/projects/-flux/docs/mom.ps.gz.

[22] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann, "Interface and execution models in the Fluke kernel," in *Proceedings of the First Symposium on Operating System Design and Implementation*, February 1999. To appear.

[23] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A flexible, optimizing IDL compiler," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1997.

[24] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler, "User-level checkpointing through exportable kernel state," in *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems*, October 1996.

[25] M. Shapiro, D. Plainfose, and O. Gruber, "A garbage detection protocol for a realistic distributed object-support system," Tech. Rep. 1320, INRIA, Nov 1990.