Performance of Generative Programming Based Protocol Implementation

By

Zheyin Li

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Master of Science
Information and System Science

School of Computer Science
Carleton University

Ottawa, Ontario

May 29, 2003

The undersigned hereby recommend to the Faculty of Graduate studies and Research Acceptance of the thesis,

Performance of Generative Programming Based Protocol Implementation

Submitted by

Zheyin Li

in partial fulfillment of

the requirements for the degree of

Master of Science Information and System Science

Dr. Frank Dehne

(Director, School of Computer Science)

Dr. Michel Barbeau

(Thesis Supervisor)

Carleton University

May 29, 2003

Abstract

Protocol Implementation Framework for Linux (PIX) is a generative programming (GP) based protocol development framework. It aims to capture the similarities in behaviors among different layers of protocols and to group solutions to crosscutting concerns of communication systems. It achieves a high degree of configurability by providing several combinations which could be chosen to generate desired protocols.

This thesis addresses the following open question: how does the performance of generative programming based protocol implementation compare with traditional protocol implementation techniques?

This thesis provides an answer to this question. A benchmark is developed to give a thorough performance analysis of PIX to contrast it with other protocol development frameworks. The benchmark compares the performance of bulk data transfer. The file transfer protocol (FTP) is used for comparison purposes. Latency, throughput and resource usage measurements are provided in order to compare the performance of generative programming based PIX with NcFTP, which uses structured programming, and x-Kernel, which uses structured and object-based programming.

This thesis also presents a performance analysis of file transfer based on the protocol stack that is exclusively developed from PIX (GP) over packet radio.

Acknowledgment

First of all, I want to thank God from the bottom of my heart for leading me through this whole long journey. I am so grateful for his love and blessings.

I would like to express my sincere gratitude and respect to my supervisor, Professor Michel Barbeau, for his crucial guidance, non-stop patience, invaluable advice and great help. I feel extremely fortunate and honored to be his student.

I would like to acknowledge School of Computer Science, Carleton University for giving me such a wonderful opportunity to study for Master of Science, Information and System Science program. Sincere thanks to technical team, especially to Mr. Gerardo Reynaga for his prompt and generous help.

I would also like to offer my appreciation to other team members who work with me on Small Satellite project, especially Mr. Song Zhang, Ms. Huan Qi, Ms. Honghui Luo, Ms. Maoning Wang and Mr. Maged Elasssar. The discussions I have had with you have always been so helpful and enjoyable. I would like to acknowledge the suggestions, encouragement and help from Ms. Jun Chen, Ms. Hua Guo, Mr. Miguel Martin Vargas and Mr. Javier Govea Mata.

I also want to thank my friends Ms. Xiaofei Lin, Ms. Wei Zhang, Ms. Nancy Liu, Ms. Yan Zhang, Mr. GuoMing Wei and Mr. Bin Zhao for bringing me so much joy.

My sister Mrs. Qin Li and brother-in-law Mr. YongTao Chen deserve special thanks for their understanding, unconditional support and firm confidence in my abilities. Last but not least, I would like to thank my parents Mrs. Xiude Liu and Mr. Hui Li for their constant love and encouragement. This thesis is dedicated to my family, especially to my newly born baby nephew, Jeffrey Yang Chen.

Table of Contents

List of Tables

List of Figures

Table of Acronyms

API	Application Interface
CSMA	Carrier Sense Multiple Access
DTP	Data Transfer Process
FTP	File Transfer Protocol
GEO	Geostationary Earth Orbit
GP	Generative Programming
GPRS	General Packet Radio Service
GSM	Global System for Mobile
IPC	Inter-process Communication
KISS	Keep It Simple Stupid
LEO	Low Earth Orbit
MAC	Medium Access Control
NASA	National Aeronautics & Space Administration
PI	Protocol Interpreter
PIX	Protocol Implementation Framework under Linux
RPC	Remote Procedure Call
TNC	Terminal Node Controller
RTT	Round Trip Time
UI	Uniform Interface
UPI	Uniform Protocol Interface

Contents

Chapter 1

Introduction

1.1 Context of the Work

The strategies for network protocol implementation can be categorized into three approaches: kernel level, user level or mixed approach. This classification is according to the placement of the implementation code relative to the operating system. Each approach has its pros and cons. The kernel level approach can achieve better performance because the input processing is handled at low level, which reduces latency. The user level approach makes code development and maintenance relatively easier, but performance could suffer. A mixed approach is a compromise of these two strategies and is only used under certain circumstances.

Implementing a network protocol from scratch is not a trivial task. A natural solution to this problem is to build a framework which contains the common components of different protocols. Developers construct application-specific protocols by reusing, extending and customizing the framework. Linux [BBD+98], BSD Unix [MBK+96] and System V Streams [AT87] are some well-known protocol implementation frameworks. Protocol Implementation Framework for Linux (PIX) [BB02] is a framework based on generative programming (GP) [CE99]. GP is a technique to build models for families of systems and then generate concrete and customized systems from these models. With the aid of GP, PIX targets supporting families of protocols. PIX captures common protocol behavior models and groups solutions to the cross-cutting concerns of different protocols so that developers can assemble various protocols to satisfy different needs. Commonalities and variations between protocols are captured as features, or in other words, properties, of the *Protocol* abstract object. By feeding the specification of these features to PIX, its protocol generator automatically produces base classes for the required protocol providing the core of the behavior. Currently, protocols constructed from this framework are at the user level.

The design of protocols in PIX is inspired by the x-Kernel object-based protocol development framework [HP91]. PIX reuses some of the key concepts of x-Kernel, such as *Protocol* and *Session*. PIX and x-Kernel share similar concepts, while the design goals and implementation techniques are quite different. The fundamental distinction between PIX and x-Kernel is that PIX is designed to obtain a high degree of configurability – it currently provides 48 combinations of ways to assemble a protocol. Providing configurability is not only the primary difference between PIX and x-Kernel, but also between PIX and the other aforementioned protocol development frameworks. The Horus and Ensemble [Bir00] framework provides configurability by means of enabling assembling stacks of micro-protocols to fulfill needs for group communications. The configuration is based on the functionality of targeted protocols. In contrast, PIX emphasizes protocol structure and optimization instead of functionality. The protocols generated from PIX stem from a well-defined prototype and specific functionalities need to be added to this prototype in order to make it a complete network protocol.

1.2 Motivation

To the best of our knowledge, PIX is the first attempt to use GP in network protocol development. PIX was designed to make protocol development work easier, but not at the expense of correctness or performance. Zhang [Zha02] provides a partial performance evaluation of PIX by comparing the choices that PIX offers for the *Message* and *EventManager* features. However, the performance of PIX and GP based protocol implementation, with respect to other traditional software development approaches, still remains an open question. Such a performance analysis is a necessary task.

1.3 Thesis Contribution

This thesis contributes to the evaluation of the performance of PIX and GP based protocol implementation by analyzing the performance of implementations of the file transfer protocol (FTP) as the core performance benchmark. FTP is an application protocol that sits at the top of the protocol stack. Since the message flow passes through the entire stack, FTP gives a picture of the overall performance of the complete system. Furthermore, latency and throughput of file transfer can be measured. These two parameters are key indicators of the performance of a network.

The main contributions of the thesis are as follows:

- Performance comparison of a PIX based FTP (FTP-PIX) with an x-Kernel based FTP (FTP-x-Kernel) and Linux NcFTP over Internet sockets. First, an FTP implementation is developed from PIX and x-Kernel frameworks respectively, then an Adapter protocol is developed to ensure a uniform transmission base. The latency, throughput and resource utilization status with respect to CPU time and memory footprint of FTP-PIX, FTP-x-Kernel and NcFTP are provided.

- Performance analysis of a PIX based protocol stack over packet radio communication.

From the analysis of testing results, we also identified a performance bottleneck in the *Map* library of the current PIX implementation in the case that one session receives multiple packets back-to-back. We provided a solution by adding cache capability. We also demonstrated from experiments that this enhancement noticeably improved the performance in the case of one network host continuously receiving data units from the same remote host.

The second part of the major contribution is contained in the paper [EZB$^+$03] that has been accepted by 17th Annual AIAA/USU Conference on Small Satellites.

1.4 Road Map

This thesis is structured in the following manner: Chapter 2 reviews and characterizes related work on building communication protocols from different approaches. Emphasis is placed on two protocol development frameworks, x-Kernel and PIX. Chapter 3 introduces the file transfer mechanism and its performance analysis criteria. Chapter 4 gives an overview of the design and implementation of FTP based on PIX and x-Kernel respectively. It also covers how the *Map* library was enhanced to solve the potential performance bottleneck. Chapter 5 gives a quantitative performance evaluation of the FTP-PIX, namely in latency, throughput and resource usage by comparing performance with FTP-x-Kernel and NcFTP. Chapter 6 presents a performance analysis of protocol stack exclusively developed from PIX over packet radio transmission. Chapter 7 concludes the thesis and identifies opportunities for future work.

Chapter 2

Overview of Protocol Development Strategies

This chapter serves as a survey of existing protocol development strategies, focusing on framework-based approaches. We start by introducing protocol development framework and describing why it is useful in Section 2.1. Section 2.2 goes through several kernel level approaches by initially introducing Linux and FreeBSD as two network protocol development environments, and then describing the Socket Interface and System V Streams, which are kernel level protocol development frameworks. In Section 2.3, we present some user level approaches, including microkernel operating system and two typical examples of protocol development frameworks: x-Kernel and PIX. In addition, the Horus and Ensemble project, as a work related to PIX, is also briefly discussed. Section 2.4 discusses the hybrid approach, which combines the user level and kernel level approaches. In Section 2.5, we briefly discuss the strategy of network protocol development through programming language support. Finally, we give a short summary in Section 2.6.

2.1 Protocol Development Framework

Strategies for network protocol development can be categorized into three different approaches based on their positions relative to the operating system: kernel level, user level and a hybrid of both. Building a network protocol from scratch is a non-trivial task using any of these approaches. Based on the fact that protocols share similar architectural and behavioral model across layers, protocol developers always attempt to find solutions to capture the common components from different protocols in order to facilitate future protocol development. One form of these solutions is protocol development framework.

Generally, a protocol development framework is comprised of two parts: 1) a set of structural guidelines, which determine protocol implementation details, such as the interface among protocol modules or layers; and 2) a set of library routines to perform common protocol functions. Protocol development framework has been used in both kernel level and user level approaches, for example the System V Streams [AT87] in kernel level approach, and x-Kernel and PIX in user level approach.

Protocol development framework has the following advantages:

- ❖ Code reuse: This claim is twofold. The first is the reuse of provided routines, which could be used by all protocols in the framework. The second is the reuse of a given protocol implementation, since a uniform protocol interface allows it to be flexibly composed of different adjacent protocols in different context;
- ❖ Consistency: The consistent structure and interface imposed on protocols makes it easier to develop new protocols and maintain or modify existing protocols;
- ❖ Efficiency: Performance of protocols in the framework is promoted by a protocol structure designed for efficiency and use of carefully tuned subroutines.

2.2 Kernel Level

With the kernel level approach, the network code is placed in kernel space and applications communicate with the operating system through a certain interface; the most popular one is socket interface deployed by FreeBSD, Linux and a lot of Unix based OSs. All network processing is hidden from the user and performed inside the kernel. In this way, high performance is achieved when application data do not need to cross the frontier of user and kernel boundary. Furthermore, network header data only need to be transferred from kernel to the receiver's address space so that reduces data copying [MB92]. It is particularly efficient for network layer and link layer protocols, which involves large amount of data and complicated control flow. However putting network protocol in kernel is a tedious and error-prone task, as code maintenance and debugging could be very frustrating under certain circumstances.

2.2.1 FreeBSD

FreeBSD [MBK$^+$96] is an advanced operating system for x86 compatible, DEC Alpha, and PC-98 architectures. It is derived from BSD Unix, the version of Unix developed at the University of California, Berkeley. FreeBSD offers advanced networking, performance, security and compatibility features, such as support for IPsec which allows improved security in networks, and IPv6, which is the next-generation of Internet Protocol. Network protocols are implemented as a separate software layer logically below the socket interface in the kernel. The kernel provides many ancillary services, such as buffer management, message routing, standardized interfaces to the protocols and interfaces to the network interface drivers for the use of the various network protocols. Network optimizations such as zero-copy sockets and event-driven socket input/output (I/O) are working in progress under FreeBSD.

2.2.2 Linux

Linux, like Unix, is a multi-user, multi-tasking, multi-processing operating system. It is widely used for routers, gateways, client and servers, as well as in embedded systems. Linux provides an implementation for multiple protocols at kernel level; for example, the dominating protocols are collected under the TCP/IP suite for the communication via Ethernet. Through Serial Line Interface Protocol (SLIP) and Parallel Line Interface Protocol (PLIP), computers link together via serial and parallel interfaces. Amateur X.25 (AX.25), 802.11 and Bluetooth provide a way of communication using radio signals. Other protocols include Internetwork Packet Exchange (IPX) realized by Novell, AppleTalk for transmission of Apple data [BBD$^+$98] and *etc.* When a process communicates via the network, it uses the functions provided by socket interface as discussed below.

2.2.3 Socket Interface

Socket is the de facto industry standard interface for writing network applications that are built on top of the network protocols hidden in kernel space. It is more frequently introduced as the common way to write network applications, rather than network protocols. However it is still discussed here in relation to network protocol development, since FTP, which could be treated as a network application protocol, is typically developed using socket interface under Linux or Unix systems.

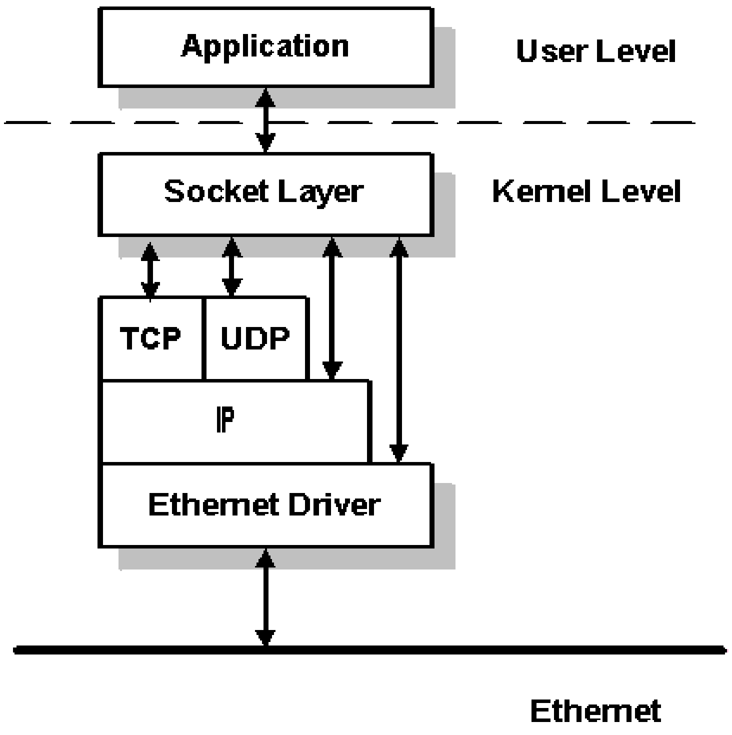

Figure 2.1 Network architecture using socket under Linux.

Socket represents a unique communication channel between two hosts, which allows applications to connect to a remote computer, to send or receive data, and to listen to incoming connections. It consists of a series of system and library calls, header files, and data structures [BBD$^+$98]. Applications can access kernel-resident networking protocols, such as the TCP/IP suite, through socket system calls. Applications can also use socket library calls to manipulate network information, for example, to map service names to service numbers, or to translate the byte order of incoming data to make it appropriate for the local system's architecture. Sockets can take several forms, according to the service provided. Most widely used are: 1) stream socket for reliable connection oriented service, such as TCP; 2) datagram socket for connectionless service, such as UDP; 3) raw socket used to access network layer protocols, such as IP; and 4) packet socket, which is used to receive and send raw packets right above the link layer, for example, Ethernet. Figure 2.1 illustrates the architecture of network protocol stack using socket interface. In this architecture, the message crosses the dividing line between user-space and kernel-space at a very early stage (at the system call layer). This makes debugging the network stack very difficult.

2.2.4 System V Streams

In System V Streams [AT87], protocols are also placed in kernel space. It was originally designed to support character I/O and later extended to support protocols. It consists of system calls, kernel routines, and kernel utilities which are used to implement everything from networking protocol suites to device drivers. It defines a processing model and a standard interface for character I/O within the kernel and between the kernel and the rest of the system. This interface is block-oriented: all parameters of an operation, including the identity of the operation itself, are buffered in a block that is passed into the protocol module. Network protocol stacks can be built and modified using the streams architecture, by writing modules based on a standard interface and then plugging them dynamically into the kernel's data route. Figure 2.2 displays the architecture of Stream.

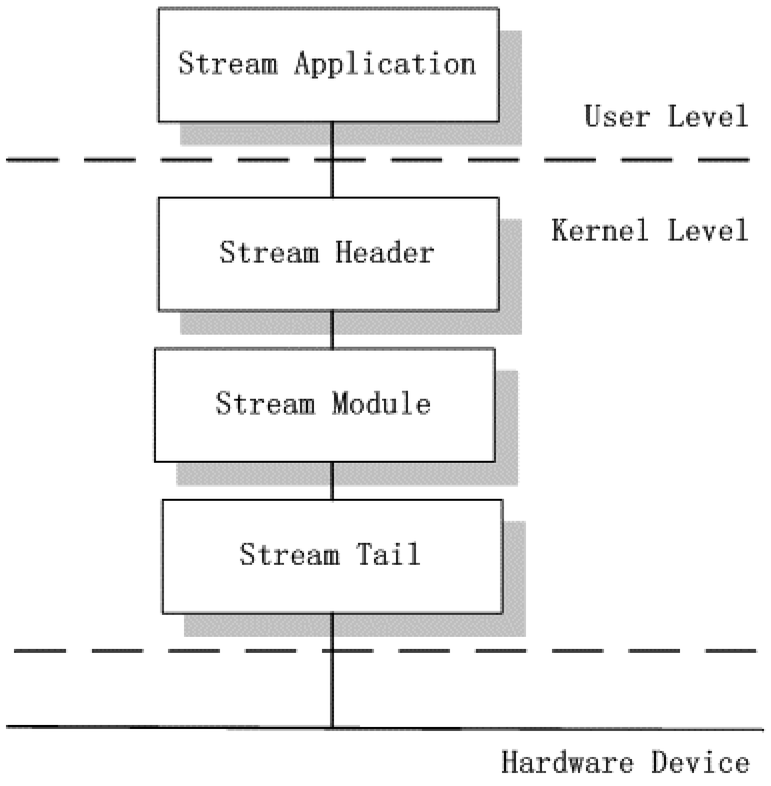

Figure 2.2 Network architecture with System V Streams.

A *stream* is defined as a full-duplex data path between a process in user space and a streams device driver in kernel space. In general, a stream can be visualized as a chain of processing elements that pass messages up and down from a user process to a driver or module in kernel space. The end of a stream nearest to the user space is called the *stream head*. It provides the interface between user and kernel space. It is essentially responsible for copying messages from user space into kernel space and passing them downstream, and vice-versa. The bottom element, the *stream tail*, can be one of the following three elements: a hardware device driver (*e.g.* for a network interface card), a pseudo-driver which provides services to applications (*e.g.* terminal emulator), or the other end of a streams-based pipe. In between the stream head and the stream tail can be any number of stream processing modules. The stream head and the stream tail are mandatory while stream modules are optional components in a stream.

To compare with the popular usage of socket, there are also many commercial Unix distributions (Sun Microsystems SunOS/Solaris, Hewlett Packard HP-UX, SGI Irix, IBM AIX, SCO Unixware, DEC Digital Unix) incorporating the features of System V Release 4.0, and thus including native streams support.

2.3 User Level

The user level approach means a protocol stack runs as one or more user processes. Moving network stack to user level gains the following main benefits:

- ❖ Shortens the code revise/test cycle: Kernel level code development usually includes an additional step in the code revise/test cycle: system reboot. This inconvenience increases the turnaround from a few seconds to a few minutes.
- ❖ Eases code maintenance and debugging. Modification to the code can be compiled separately without compiling the whole system, and the code can be debugged without kernel level debuggers, which usually only supports assembly language and arrests the whole system during a debug session.
- ❖ Improves code stability. An unstable protocol stack will only affect the application using it, and is unlikely cause catastrophic system crash.
- ❖ Enhances code portability. Kernel releases usually vary significantly from version to version. Therefore network protocol implementation built into kernel space often necessitates changing of kernel code or structures (*e.g.* the Mobile IPv4

project carrying on at Helsinki University [Mal00]). In the mean time, it requires substantial effort to transplant the code developed in one kernel version into another version.

Besides the advantages listed above, protocol development at the user level also achieves the benefit of customization and more manageable layer control. In terms of the performance concern from the critics, Maeda and Bershad [MB92] argued that the primary determinant of performance is the structure of a protocol implementation rather than the location.

Microkernel operating systems, such as Mach, QNX and VxWorks put the protocols in user space. The x-Kernel and PIX frameworks that this thesis has been working with also belong to the user level category.

2.3.1 Microkernel Based Operating Systems

The microkernel architecture abstracts lower-level OS facilities, implements them in kernel space and moves higher-level facilities to processes in user space. Usually what distinguishes higher from lower-level OS facilities is whether or not they can be implemented in a platform independent manner. In microkernel architectures, device-drivers, virtual memory, process/task/thread management/scheduling, and other such facilities are implemented in the kernel, while file systems and networking which employ the lower-level facilities provided by the microkernel are implemented in user space.

The microkernel generally relegates protocols to a specific user address space as a user process. User level applications indirectly access the address space by passing requests and replies to each other through an inter-process communication (IPC) provided at kernel level.

2.3.2 x-Kernel

The x-Kernel [PD96] is a full-fledged protocol framework that was first developed in the Network Group, at the University of Arizona. It stems from an interest in how the structure of an operating system influences protocol performance. It began as an operating system, and then evolved to become a networking subsystem which can be installed in other operating systems. The x-Kernel integrates three ingredients: 1) it defines a uniform set of abstractions of encapsulating protocols; 2) it structures the abstractions in a way that makes the most common patterns of interaction efficient; and 3) it supports primitive routines that are applied to common protocol tasks. Although implemented in C programming language, the infrastructure enforces a minimal object-oriented style; in other words, each object supports a uniform set of operations.

There are three main abstract objects defined in the x-Kernel: *Protocol*, *Session* and *Message*.

Protocol performs two functions. First, it implements the algorithms that make up a piece of communication functionality. For example, an IP protocol object would add and strip IP headers, and process the headers to route messages to the right destinations. Second, it is responsible for managing sessions, which represent certain connections. To be more specific, the protocol will create sessions, keep track of each session, demultiplex messages to particular sessions and close sessions. Protocols are connected into a protocol graph at configuration time. Protocols invoke each other by launching the operations defined as Uniform Protocol Interface (UPI). Executing some scripts automatically generates the initialization code for each protocol that has been configured as part of the graph. Figure 2.3 gives an example of protocol graph. Remote Procedure Call (RPC) will use UDP as its transport layer protocol, which connects to IP, which connects in turn to Ethernet Protocol (ETH).

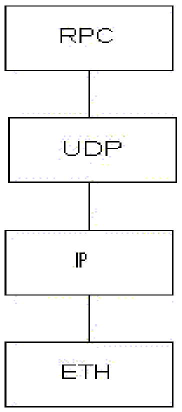

Figure 2.3 An example of x-Kernel protocol graph.

Session is an instance of a *Protocol* for a particular network connection, which holds information for a connection. It is created at runtime either actively, when the application initiates the communication, or passively, when a higher level protocol informs its lower protocol of its readiness to accept connections. In the former case, a new session object is created right away while in the latter case an enable object is first created to remember the host's willingness, and then real sessions are created to handle the new connections initiated by a remote network host.

Message is a two-part structure that separates the header from the user data. There are two basic network characteristics to network message processing [Mos96[1]], prepending header to outgoing messages and stripping header from incoming messages. A buffer tree manager is utilized to handle message data.

The relationship between the protocol and session in client side is illustrated in Figure 2.4. Intuitively, a high-level protocol (hlP) invokes the *open* operation of a low-level protocol (llP) to create an active session (llS). This session is created and maintained by the llP on behalf of the hlP. The llS and its identifier are stored in protocol's active map. Sending out a message is achieved by calling the *push* operation on the session (hlS, llS), in which headers are added, the message may be fragmented into multiple message objects, or the message may suspend itself while waiting for a reply message. When a message arrives, the job of *demux* (hlP, llP) is to find the pointer to the right session (hlS, llS) through the identifier embedded in the message and *pop* the message to this session (hlS, llS) for processing. The case in server side is a bit different from client side as the sessions are passively opened. Usually, the hlP informs the llP that it is ready to receive connection by invoking llP's open_enable operation. When a message arriving from the network, the llP call the hlP's open_done operation to inform the hlP that it has created a session on its behalf.

The x-Kernel provides management services, like mapping and events, to the developers to relieve them from having to implementing their own strategies. A map manager [Mos96[2]] helps to set up mapping between identifiers in messages and session objects. An event manager is implemented to handle events such as timeouts. These services are highly optimized to execute in the x-Kernel environment.

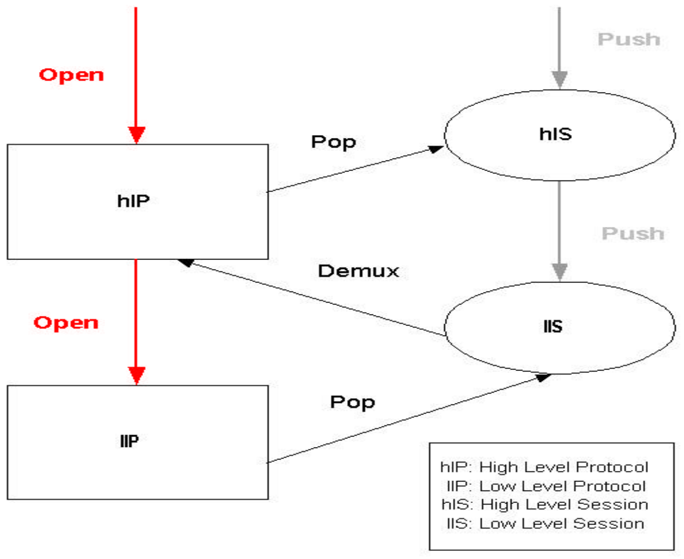

Figure 2.4 Protocol and Session interactions in x-Kernel.

2.3.3 PIX

PIX is designed to capture similarities in behaviors among different layers of protocols and to group together the solutions to cross-cutting concerns during protocol implementation. This allows developers to assemble various abstract protocols from basic components to fulfill different needs. It is based on GP, which addresses the development of a family of systems and belongs to domain engineering. Domain engineering considers development for reuse and consists of three basic steps [CE99]. Firstly, domain analysis involves domain scoping and feature modeling. Domain scoping determines which system, belongs to the domain and which are not. Feature modeling identifies the common and variable features of the domain concepts and dependencies among the variable features. In this phase, a feature diagram is depicted to reveal the variability contained in the domain space. Secondly, domain design produces a common architecture for the family of systems. Thirdly, domain implementation implements components, generators and any reusable infrastructure.

A feature diagram of PIX is drawn as Figure 2.5, according to [BB02]. *Protocol,* which is the root in the diagram, is composed of several features, namely (from left to right in the diagram) Participant, Message, UI, Session, EventManager, PassiveOpen and ActiveOpen. Features can be mandatory (filled circle at the end of an edge) or optional (open circle at the end of an edge), and an arch between the features shows an alternative relationship. Using Figure 2.5, it is possible to calculate that protocols can have 48 different variations (two alternatives for Message, two for Session, four for EventManager and three for protocols).

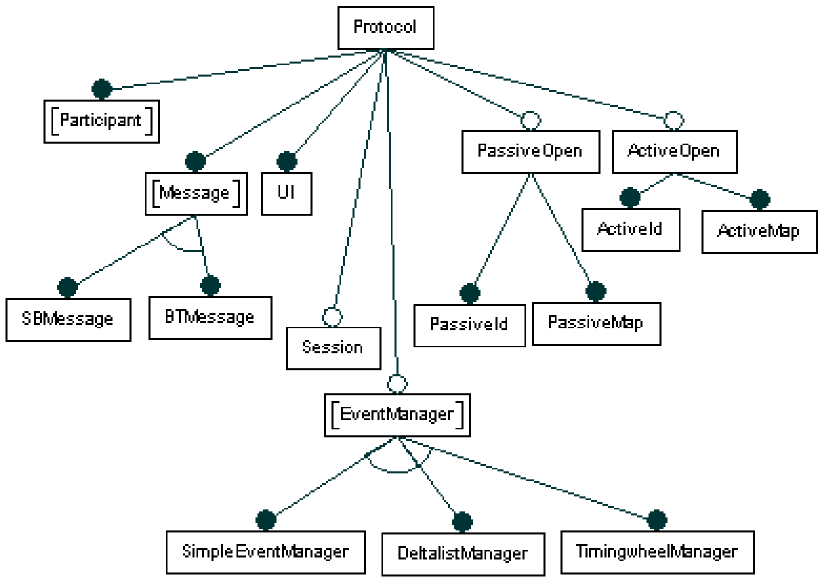

Figure 2.5 Feature diagram of PIX.

Protocol, Session and Message have the same meaning as under x-Kernel, except that Message provides two alternatives: SingleBufferMessage and BufferTreeMessage. A detailed description of other features in the diagram is listed as follows:

- *Participant* identifies the participants in the network that communicate with each other.
- *EventManager* provides a mechanism for scheduling a procedure to be called after a certain amount of time. By registering a procedure with the event manager, protocols are able to timeout and act on messages that have not been acknowledged or perform periodic maintenance functions. This diagram shows three variations: SimpleEventManager, DeltaListManager and TimingWheelManger.
- *Uniform Interface (UI)* defines the common operations to all protocols and sessions. Each protocol has its own specific implementation of UI.
- *ActiveOpen* is required when a node (like a client) knows the other nodes it wishes to communicate with. It has two sub features, *ActiveId* and *ActiveMap.* *ActiveId* is the structure containing the identifiers of both the local and the remote participant, and serves as the key in *ActiveMap*, which is a map containing the bindings between the *ActiveId* and the corresponding active sessions.
- *PassiveOpen* is required when a node (such as a server) is willing to accept connections from remote nodes. An enabled object is created to remember the fact that this node is willing to access connections. It also has two sub features: *PassiveId* and *PassiveMap*. *PassiveId* contains the local participant and acts as the

key in *PassiveMap*, which stores bindings between *PassiveId* and the corresponding enabled objects.

The next step in the design cycle of domain engineering is domain design, in which, a common architecture of the family of protocols is developed. Features are also mapped to components in this stage. GenVoca grammar, a widely used methodology to capture the system architecture in a hierarchical layered fashion, is utilized. Figure 2.6 gives the Gen Voca representation of PIX architecture in seven layers. The Message and EventManager features of the Session concept are collected together in the bottom layer. Vertical bars separate the alternatives provided by Message and EventManager. This layer serves as the configuration repository to define the SessionUI layer (L6), which reflects the Session object. ProtocolUI configuration repository (L5) requires three features: Message, Session, and Participant. This layer is used to define the layer above, ProtocolUI, which is the abstraction of an interface to a protocol. Layer three (L3) is the configuration repository for a protocol definition and includes a dependency on the ProtocolUI layer and all additional features. The PassiveOpen feature depends on the ActiveOpen feature as shown in layer two (L2). The top layer Protocol groups three different alternatives that can be taken to define a protocol. PassiveOpen with PassiveOpenWithAO provides a basis for protocols that have sessions with both client and server behavior. ActiveOpen parameterized with Config provides a basis for a protocol with sessions that only acts as a client. And the third option ProtocolUI is used by protocols that do not embed sessions.

```
Protocol : (L1)
      PassiveOpen[PassiveOpenWithAO] | ActiveOpen[config] | ProtocolUI
PassiveOpenWithAO : (L2)
      ActiveOpen[Config]
Config : (L3)
     ProtocolUI EventManager ActiveId ActiveMap PassiveId PassiveMap
ProtocolUI: (L4)
     Protocol[ProtocolUIConfig]
ProtocolUIConfig: (L5)
     Message | SessionUI | Participant
SessionUI: (L6)
      Session[SessionConfig]
SessionConfig: (L7)
      Message:  SingleBufferMessage | BufferTreeMessage
      EventManager:SimpleEventManager | DeltalistManager |
                   TimingwheelManager
```

Figure 2.6 Gen Voca representation of PIX architecture.

In the last step, domain implementation turns components to C++ classes. A component from a given layer takes a component from the layer below it as its parameter, which means components are implemented as parameterized classes, or to be more specific, are represented as C++ class templates. Eventually, the Protocol class template is coded as having a parameter that serves as the configuration repository containing the values to the features of the wanted protocol. There are two ways to assemble a protocol: manual assembly and automatic assembly. In manual assembly, the developer writes his/her own configuration repository for each system in this family, which is tedious and restrictive. A more efficient and practical way is the auto generation of systems by providing their specifications to a generator. A generator is a class template that encapsulates the rules defining how components can be assembled together. In general it performs the following operations: it validates if the system could be built, completes the specification (by applying default) and assembles the implementation components [CE99]. A detailed explanation of how the protocol generator is implemented can be found in [Zha02]. Figure 2.7 gives an example of a specification to generate a protocol from protocol generator. It generates an FTPBase protocol which conforms to common protocol interface *ProtocolUI*, using *SimpleEvent* as event manager, supporting both active open (*with_activeopen*) and passive open (*with_activeopen*) capabilities from the protocol generator. By adding FTP capabilities to the generated protocol prototype FTPBase, we get the functional FTP, which is discussed in Chapter 4.

```
typedef PROTOCOL_GENERATOR
<
ProtocolUI,
FTPActiveId,
FTPActiveMapConfig::FTPActiveMap,
FTPEnable,
FTPPassiveId,
FTPPassiveMapConfig::FTPPassiveMap,
FTPHeader,
SimpleEvent,
ProtocolUI,
with_activeopen,
with_passiveopen
>::RET FTPBASE;
```

Figure 2.7 A specification example provided to protocol generator.

Table 2.1 summarizes and compares the characteristics of these two frameworks. From this table, we can see that, PIX has superiority in having better structure and providing more choices in event and message handling, while its Map utility lacks caching capability. This could create a potentially serious performance bottleneck in the current implementation of PIX. We are going to discuss this issue more thoroughly in Chapter four.

	X-KERNEL	PIX
Implementation Approach	Object-based	Object-oriented Generative Programming
Implementation Language	C	C++
Event Manager	TimingWheelManager	SimpleEventManager, DeltaListManager, TimingWheelManager
Message Manager	BufferTree Message	SingleBufferMessage, BufferTreeMessage,
Map	Hashtable with Cache	Hashtable without Cache

Table 2.1 Comparison between x-Kernel and PIX.

2.3.4 The Horus and Ensemble Project

From the previous section, we know that, PIX is configurable by providing 48 ways of generating generic protocols. Here we are going to briefly introduce two other frameworks: Horus and Ensemble [Bir00]. They provide configurability by means of configuring functional-oriented simple protocols to fulfill the mission of a complex protocol. Horus is a portable group communication subsystem. It seeks to simplify the monolithic structure of communication systems by showing how complex protocols can be broken into simple micro-protocols and stacked to match the needs of a particular environment. Ensemble is rather similar to Horus, although rewritten using a high level programming language O'Camel. For an application builder, Ensemble provides a library of "tiny" protocols that can be used to quickly build complex distributed applications. Each "tiny" protocol implements several simple properties, such as total ordering, security, virtual synchrony, *etc*. Individual layers can be modified or rebuilt to experiment with new properties or change the performance characteristics of the system. The application registers as many as 10 or so event handlers with Ensemble, and the Ensemble protocols handle the details of reliably sending and receiving messages, transferring state, implementing security, detecting failures, and managing reconfigurations in the system.

These two frameworks give design flexibility to developers. The limitation is that while a library of predefined layers is provided, to add or modify these would be non-trivial due to their low-level nature.

2.4 Hybrid Approach

In the hybrid approach, part of the implementation code exists in kernel space and part of the code exists in user space. Several mobile IPv6 implementations have adopted this approach, such as the one in the National University of Singapore [NUS] and in [Xu01]. In their designs, the module in the user space is responsible for controlling message while the module in kernel space handles mobile IPv6 traffic management. The performance might be better than that of pure user-level approach, since there are not many control messages, which reduces the data volume that needs to be transferred between user space and kernel space.

Apparently, this approach is a compromise between the other two. It is not widely deployed because it is hard to separate within one protocol those parts that involve high volume of data transfer from the parts that require minimal data processing. Therefore, it only suits certain special cases. To the best of our knowledge, the mobile IPv6 implementations mentioned above are the only attempts to use this approach to develop a protocol.

2.5 Language Approach

So far, the network protocol implementations have all been introduced from the system support point of view; however, there are other perspectives too, such as the implementation from a language approach [Ab93]. The language approach refers to the building of protocols through modifying existing or creating new programming languages. This approach provides high level abstractions for protocols that make requirements more precise for everyone involved in the design and implementation process. This strategy essentially embeds a protocol framework in a programming language. This approach has two main phases: 1) the communication abstractions are added to the language and 2) the support of the language is used extensively. The communication abstractions become new abstract data types and are added to the language. They can be manipulated like all other existing ones. Considerable effort has been put on this approach as well, like Morpheus [Ab93], which was developed by the same network research group as x-Kernel at the University of Arizona or FoxNet as part of Fox Project.

2.6 Summary

This chapter surveys related work on protocol development strategies. There are different ways to categorize these approaches: either according to the placement of the code relative to the OS space, or by providing specific language constructs for building protocols. Each of these approaches has a different philosophy to cope with different system requirements.

In this thesis, we are more interested in protocol development frameworks at user level, specifically the x-Kernel and PIX frameworks. Accordingly, we illustrated their structures in detail and compared their characteristics. In Chapter four and Chapter five we will discuss how an FTP is developed in these two frameworks and give a performance analysis.

Chapter 3

File Transfer Protocol and Performance Criteria

In this chapter, we provide an overview of the FTP, and then review several popular FTP implementations. Finally, we explore the performance measurement and evaluation criteria.

3.1 FTP Overview

FTP has long been the standard for moving files between hosts on TCP/IP internetworks [PR85]. It provides a mechanism to transfer data reliably and efficiently between two different computers and shields users from the variations in the file storage systems between the hosts. In this section, we illustrate the model FTP is based on, the TCP connections it utilizes, its commands and replies, and a typical usage scenario.

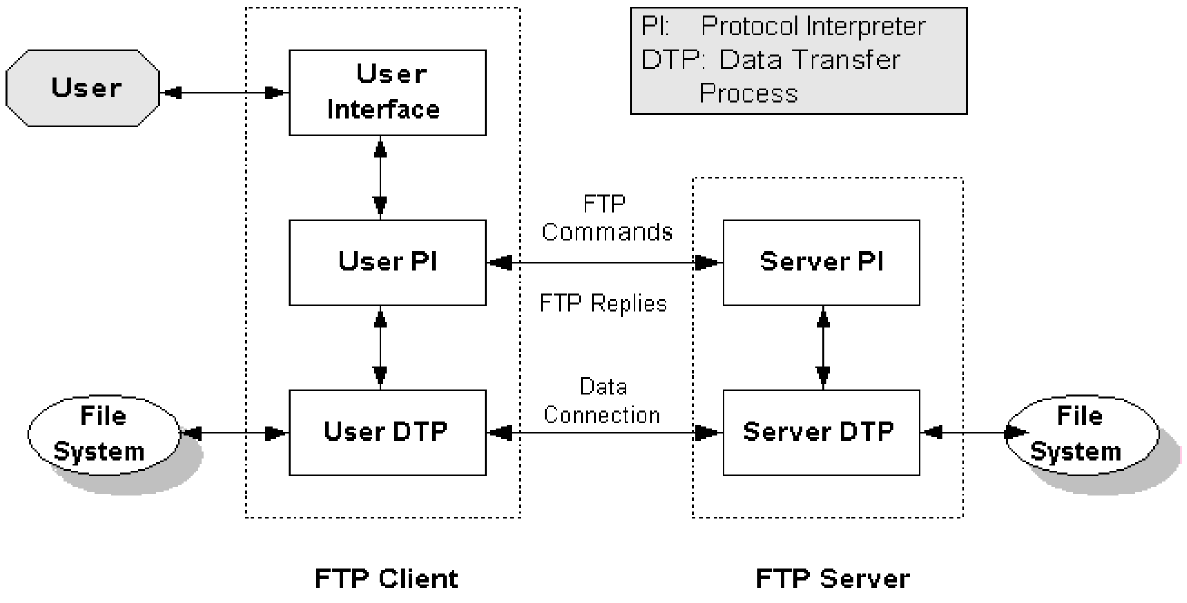

Figure 3.1 FTP client and server model.

➢ **FTP Model**

FTP follows a typical client-server model as many other TCP/IP applications. Figure 3.1 presents the FTP model. An FTP client is made up of three pieces, namely, the user interface, user protocol interpreter (PI), and user data transfer process (DTP). A user interacts with FTP client through the user interface. The user-PI translates the commands issued by the user to the functional requests understood by FTP server. The user-DTP "listens" to the data port for a connection from an FTP server process. On the server end, there is a listener process (also known as a daemon) that interprets the request from the client. The FTP daemon consists of a server-PI and a server-DTP. The former receives standard FTP commands from the user-PI, sends replies, and governs the server-DTP. The server-DTP, in its normal active state, establishes the data connection, sets up parameters for transfer and storage, and transfers data on command from its PI. The server-DTP can also be placed into a passive state, to listen to, rather than to initiate, a connection on the data port.

➢ FTP Connections

The TCP connection between the user-PI and server-PI is known as a *control connection*, which is responsible for sending commands from the client to the server and for the server returning messages to the client. The *data connection* between user-DTP and server-DTP is spawned for the transfer of data, which includes files being sent from the client to the server or from the server to the client, as well as directory information from the server to the client. The usage of separate connections for control and data offers two major advantages: 1) the two connections can select different appropriate qualities of services, for example, minimum delay for the control connection and maximum throughput for the data connection, and 2) it eliminates the trouble of embedding control commands into the data stream. There are three transmission modes in FTP: 1) stream mode, which passes data with little or no processing; 2) block mode, which formats the data into blocks, facilitating file transfer control and management and 3) compressed mode for efficient transfer. In stream data transfer mode, the termination of data connection also indicates that the end of file has been encountered.

➢ FTP Commands and Replies

FTP has a standardized set of commands, which are typically three or four characters long and are followed by an argument. The response message sent by the server consists of a numeric code followed by a text message. The success or failure status of the command can be determined from the numeric code. The text message provides more detailed information in addition to the numeric code.

➢ **Typical Usage Scenario**

In normal Internet operations, the FTP server listens on a well-known port numbered 21 for control connection requests. The client sends an ID and password to this port to gain access to the server's system. Then the client issues commands to navigate through the file directories on the server, list directory contents, and then download or upload files. When finished, the client terminates the session. It is also a standard practice for an FTP server to automatically terminate a session if there is no activity for a preset length of time. Here we offer a detailed explanation of how a file is successfully downloaded from the server in stream mode, since it is a typical and one of the most frequently used procedures.

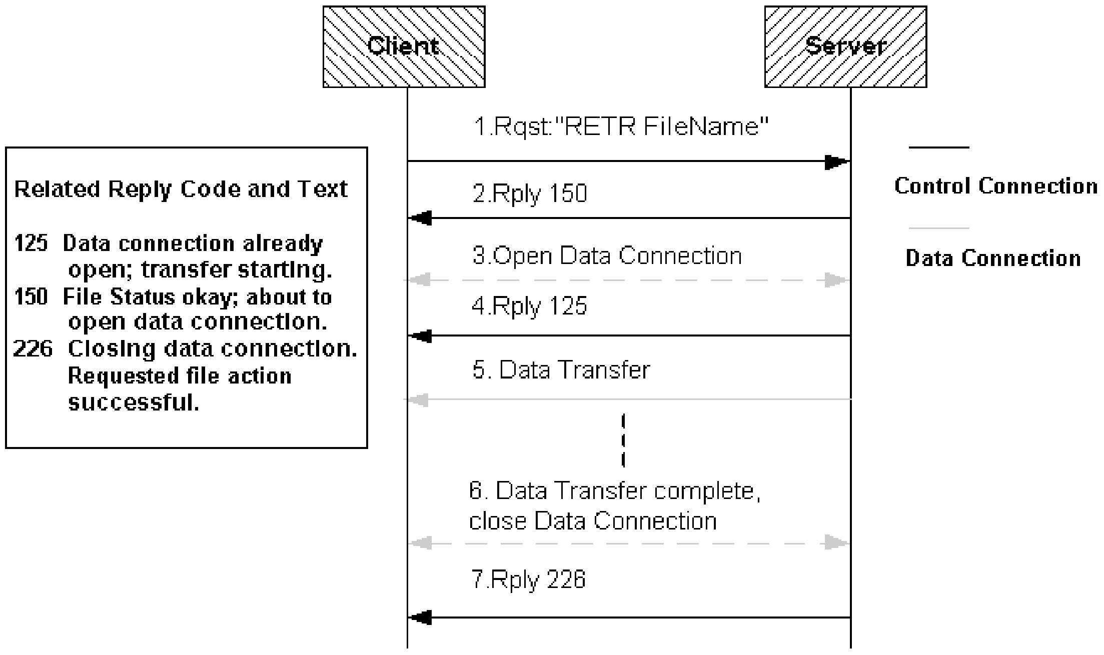
Client
Server
1.Rqst:"RETR FileName"
2.Rply 150
3.Open Data Connection
4.Rply 125
5. Data Transfer
6. Data Transfer complete, close Data Connection
7.Rply 226
Control Connection
Data Connection
Related Reply Code and Text
125 Data connection already open; transfer starting.
150 File Status okay; about to open data connection.
226 Closing data connection. Requested file action successful.

Figure 3.2 A client retrieves a file procedure.

Figure 3.2 shows the interactions between a client and a server to download a file in stream mode. First (step 1), an FTP client sends a request specifying the file name it wants to retrieve from the server side. Once the FTP server receives the request, it checks the file status, and if it is OK the server will respond (step 2) with Code 150, which tells the client the size of the file, and then opens a data connection (step 3). A confirming reply with Code 125 will be sent (step 4) to the client through control connection once the data connection is established. In the mean time, file data is sent (step 5) from the server to the client through this newly opened data connection. In stream mode, the data connection is closed (step 6) when the file data transfer is completed. Then, the server sends (step 7) a reply with Code 226 to the client communicating the result of the transfer. We may also have noticed that there are two connections, steps 1,2,4,7 take place in control connection, while steps 3,5,6 are in data connection.

3.2 Available FTP Implementations

From the first specification of FTP (RFC114) that was proposed and developed on network hosts at M.I.T in April 1971, to the formal complete specification RFC959 (updated by RFC2228 in October 1997) and RFC1579, many commands have been defined. There are still on-going efforts to add more features, and different FTP implementations vary features they support. Protocol implementers have suffered from the erroneous opinion that implementing FTP ought to be a small and trivial task. This is mistaken: FTP has a user interface, and it has to deal correctly with a whole variety of communication and operating system errors that may occur, in addition to handling a great diversity of real file systems. Here we briefly review several widely-used FTP applications under the Unix or Linux operating systems.

FTPD stands for FTP daemon, which is the classic Unix FTP server, and only supports basic FTP capability, advanced features like resuming failed transfer are absent. Recently it has been replaced by Wuarchive-ftpd [WuF], which is known as WU-FTPD. It was developed at Washington University by Chris Myers and later updated by Bryan D. O'Connor. WU-FTPD provides enhanced features such as IP-based virtual host support, transfer and command logging, on-the-fly compression, and security rules, amongst other things. Therefore, it has become the most popular FTP daemon on the Internet, used on many anonymous FTP sites all around the world. While WU-FTPD provides excellent performance and stability, it has a poor security history and it is very resource intensive. A newer version of FTP daemon, ProFTPD, emerged, and is now used by many well-known sites, including the GNU project and Linux kernel FTP resource center. ProFTPD [Pro] focuses on simplicity, security, and ease of configuration. It features a very highly customizable server infrastructure, including support for multiple virtual FTP servers, anonymous FTP, and permission-based directory visibility. TrollFTPD [Tro] is another FTP daemon, which is considerably more secure and less resource-intensive than WU-FTPD. It is intended to be lightweight and mainly used for Linux system.

NcFTP [NcF] Client is a set of FTP client application programs. The program has been in service on Unix systems since 1991 and is an alternative to the standard FTP client program. NcFTP offers many ease-of-use and performance enhancements over the operating system’s built-in FTP client, and runs on a wide variety of Unix platforms, as well as operating systems such as Microsoft Windows and Apple Mac OS. It is open source and easily modified and adapted. IglooFTP PRO [Igl] is a commercial graphic FTP client in Linux. Its main features include secure FTP connections, full transfer resume, remote directory caching, User Request Location (URL) clipboard monitoring and enhanced firewall support. The secure connection is achieved by the Secure Sockets Layer (SSL v2/v3), Transport Layer Security (TLS v1), Stanford Secure Remote Password (SRP) protocols, Message Digest (MD 4 / MD5), and Secure Hash Algorithm 1 (SHA1) encrypted passwords.

3.3 Network Performance Criteria

Network performance is measured in two fundamental ways: throughput and latency, so we discuss these two measures in this section.

3.3.1 Network Throughput

Network throughput is also referred to as bandwidth in many contexts, although there are subtle differences between these two concepts. Bandwidth is a measure of the width of a channel, in terms of its frequency spectrum. The bandwidth of the medium limits the speed at which you can communicate: the greater the bandwidth, the greater is the data rate. Since bandwidth (and some details like coding schemes, and channel noise) determines the maximum raw data rate possible, it is generally considered as the raw data rate of the channel. In this case, the unit is bits per second. Throughput is a measure of how much data can pass through a channel per unit of time. It depends fundamentally on bandwidth and efficiency of communication protocols. The efficiency of the communication protocols depends on how the protocols are designed, and how well the communicating parties make use of the raw bandwidth of the channel.

3.3.2 Network Latency

Latency corresponds to how long it takes a single bit to propagate from one end to another end in a network. The contributors to network latency include:

- Propagation Delay**:** This is simply the time it takes for a packet to travel from one place to another at the speed of light via a certain medium, for example the speed of light in free space is 3×10^8, in fiber is 2×10^8, and in copper is 2.3×10^8.

- Transmission Delay**:** The amount of time it takes to transmit a unit of data. This is a function of the network bandwidth and the size of the packet in which the data is carried.

- Other delays: This includes possible queuing, routing, and other processing and storage delays. For example, each gateway node takes time to examine and possibly change the header in a packet. Within the network, a packet may also be subject to storage and hard disk access delays at intermediate devices such as switches and bridges.

Therefore, network latency could be represented as indicated below, where Distance is the length that data will travel and SpeedOfLight is the effective speed of light over the carrier.

Latency = Propagation Delay + Transmission Delay + Other Delays (Equation 3.1)

Propagation Delay = Distance / SpeedOfLight (Equation 3.2)

Transmission Delay = Size / Bandwidth (Equation 3.3)

3.3.3 Relationship between Throughput and Latency

Throughput and latency are combined to define the performance characteristics of a given network link or channel. Their relative importance depends on different situations. Generally speaking, when sending a very small sized message, or when a message needs to traverse a huge distance, latency dominates throughput. For example, a client that sends a one-byte message to the server and then receives a one-byte message in return is in the latency bound. An application will perform much differently using a geostationary (GEO) satellite with an average round trip time (RTT) of 240-270 ms [Jam96] than it will on an across-the-room channel with 1ms RTT. In contrast, considering bulk data transfer, for example, transfer an image of several mega bytes, the more bandwidth that is available, the less time it takes to complete the transfer. In this case, throughput is of more importance.

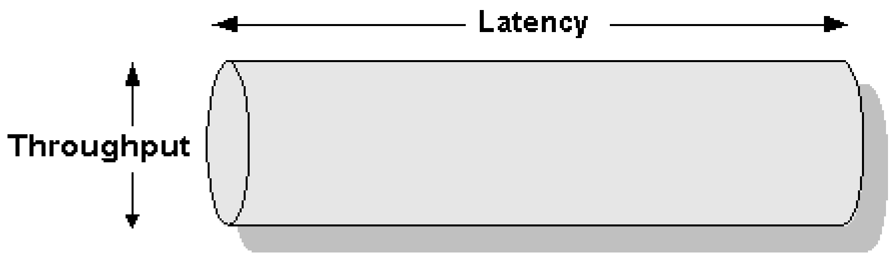

Figure 3.3 Product of latency and throughput.

By looking at the product of bandwidth and latency [PD96], we get an intuitive feel for a given channel as shown in Figure 3.3. Geometrically, this can be envisioned as a pipe. The length of the pipe is proportional to the latency of the channel. The diameter of the pipe is proportional to the bandwidth of the channel. The actual product (when the units are seconds and bits/second, respectively) is the number of bits that will fit in the channel. It is interesting because it indicates the storage capacity of the channel. If you start sending data, then the capacity of the channel tells you how many bits will be sent before the first bit even reaches the receiver.

3.3.4 FTP Latency

Based on an understanding of network latency, we further define FTP latency as how long it takes a file to be transferred from one end of the network to the other. With the help of Figure 3.2, we can take a closer look at the composition of FTP latency from the operational point of view when downloading a file. The total latency (L) is comprised of several parts: 1) the time to open and close a TCP connection for data; and 2) the time to send requests, replies and file data, which could be divided into two parts: processing delay and actual transmission time. Processing delay involves the processing time of every protocol layer. Transmission time is known from Equation 3.3, as the result of dividing Size by Bandwidth. So the variable parts of L are identified as the bandwidth of the network, the size of the file/replies/requests, and the processing time of sending messages. From a protocol point of view, the network bandwidth and messages are beyond control, so the part that varies is the processing delay, which is what we are investigating.

3.4 Resource Consumption

Resource consumption is another important performance criterion. Usually it refers to the CPU and memory usage for executing a program. It is often hard to measure since operating systems may not provide a tool to gather this kind of data. Even when a tool exists, it might not provide information of sufficient granularity in order to be used. The *gprof* tool from Linux is just one example of this. It is designed to help gather profile data to determine which functions require the most execution time [MA01]. It works like this: when a profiled executable runs, every time a function is called, its count is incremented. And *gprof* periodically interrupts the executable to determine the currently executing functions. These samples determine function execution times. Because Linux's clock ticks are 0.01 second apart, these interruptions occur at most every 0.01 second. Therefore, this tool could not provide accurate information for fast executing programs or for quickly and infrequently executing function. In our later experiments, we use *getrusage* system call and *top* command to collect information on CPU time usage and memory footprint as we discuss in Section 5.4.

3.5 Summary

FTP has now matured into a reliable, widely available standard. Essentially, all machines accessible on a TCP/IP network have working FTPs. In this chapter we briefly described FTP and the performance indicators: throughput, latency and resource usage. In the forthcoming two chapters, we will discuss the FTP implementations in PIX and x-Kernel, and then analyze the experiment results by utilizing these implementations.

Chapter 4

FTP Implementations in PIX and x-Kernel

In this chapter, we first discuss how FTP is designed and implemented in PIX, followed by the counterpart work done in x-Kernel. In the following chapter, we use these two sample implementations as experimental platforms to measure and compare the performance metrics. In order to facilitate our performance experiments, we also developed an adapter protocol in PIX and in x-Kernel respectively to serve as a uniform transmission base, which is briefly introduced as well.

In addition, during the preliminary testing, we discovered a performance bottleneck in the Map utility of the current PIX implementation. The details of the problem and a possible solution are also discussed in this chapter.

4.1 FTP Design and Implementation in PIX

In this section, we first discuss requirements of the system then show the design architecture and offer detailed explanation of classes defined for FTP in PIX.

4.1.1 System Mandate

Our objective is to measure and analyze the performance of an FTP protocol implemented in PIX based on GP. To simplify the implementation task, we only implemented a selected subset of the functionalities to fulfill the performance evaluation goal. The following is the list of commands that we implemented in our sample FTP.

- user: specify a user name
- pass: specify a password
- connect/open: establish a connection
- close: close a connection
- help: display the help information
- abor: abort the previous command
- quit: terminate a session
- size: return the size of a file
- retrieve/get: retrieve a file from the server
- store/put: store a file to the server
- list/dir: give list files in a directory
- cwd: change the working directory
- pwd: print the current working directory

4.1.2 Design

Figure 4.1 illustrates the architecture of the FTP implemented in PIX. There is a client protocol and a server protocol, each of which contains three sub layers. The top layer of FTP_Client or FTP_Server contains main functions that are responsible for initialization. The FTP_Client part is also responsible for interactions with a user. The middle layer is called an application interface (API) layer, which defines the interface for an FTP client

(Client_API) and FTP server (Server_API). In other words, the Client_API handles FTP client commands, while the Server_API consists of server handlers to these commands. The bottom layer is composed of FTPProtocol and FTPSession, which take care of real data transmission and interaction with transport layer protocols such as TCPProtocol and TCPSession. The relationship between FTPProtocol and FTPSession follows the general pattern of protocol and session interaction defined in PIX as discussed in Section 2.3.3. Briefly, a session stands for a particular network connection, and protocol manages sessions.

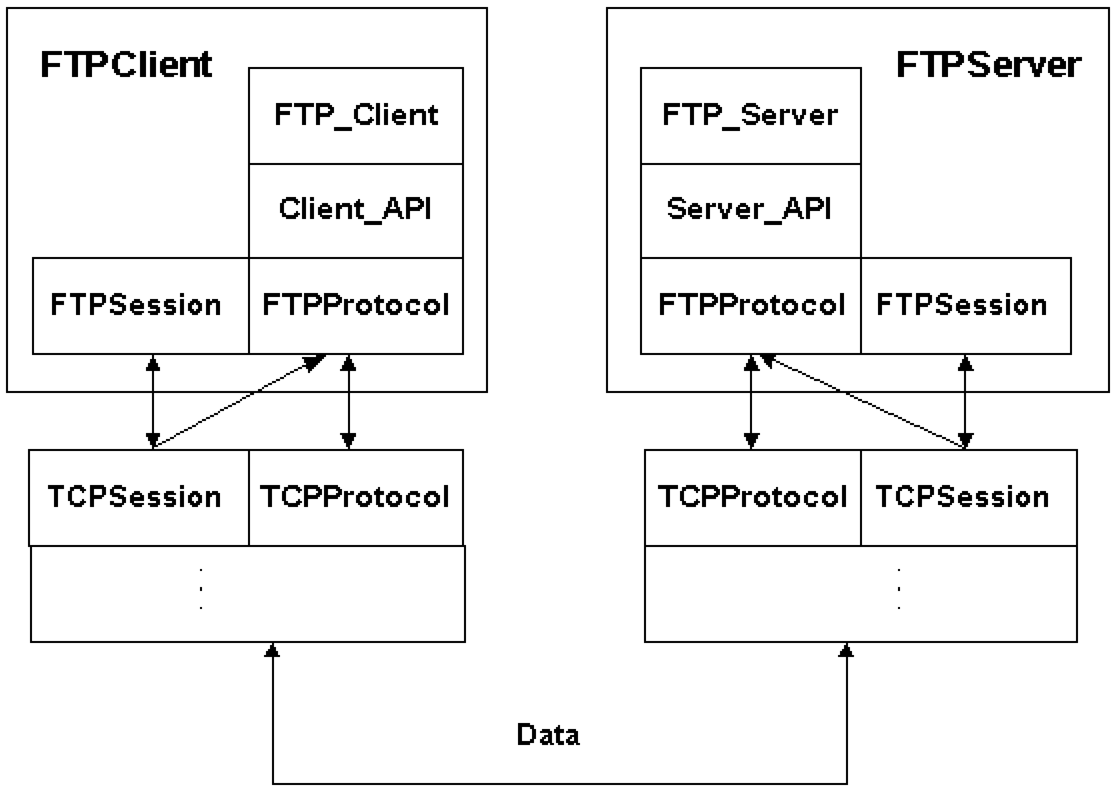

Figure 4.1The FTP architecture in PIX.

The class diagram of the PIX implementation of FTP is depicted in Figure 4.2. Each class is represented as a rectangular frame with the class name on the top, followed by class attributes in the middle and the list of methods at the bottom. The small book icons, key icons and lock icons are used to denote public, protected, and private attributes or methods respectively.

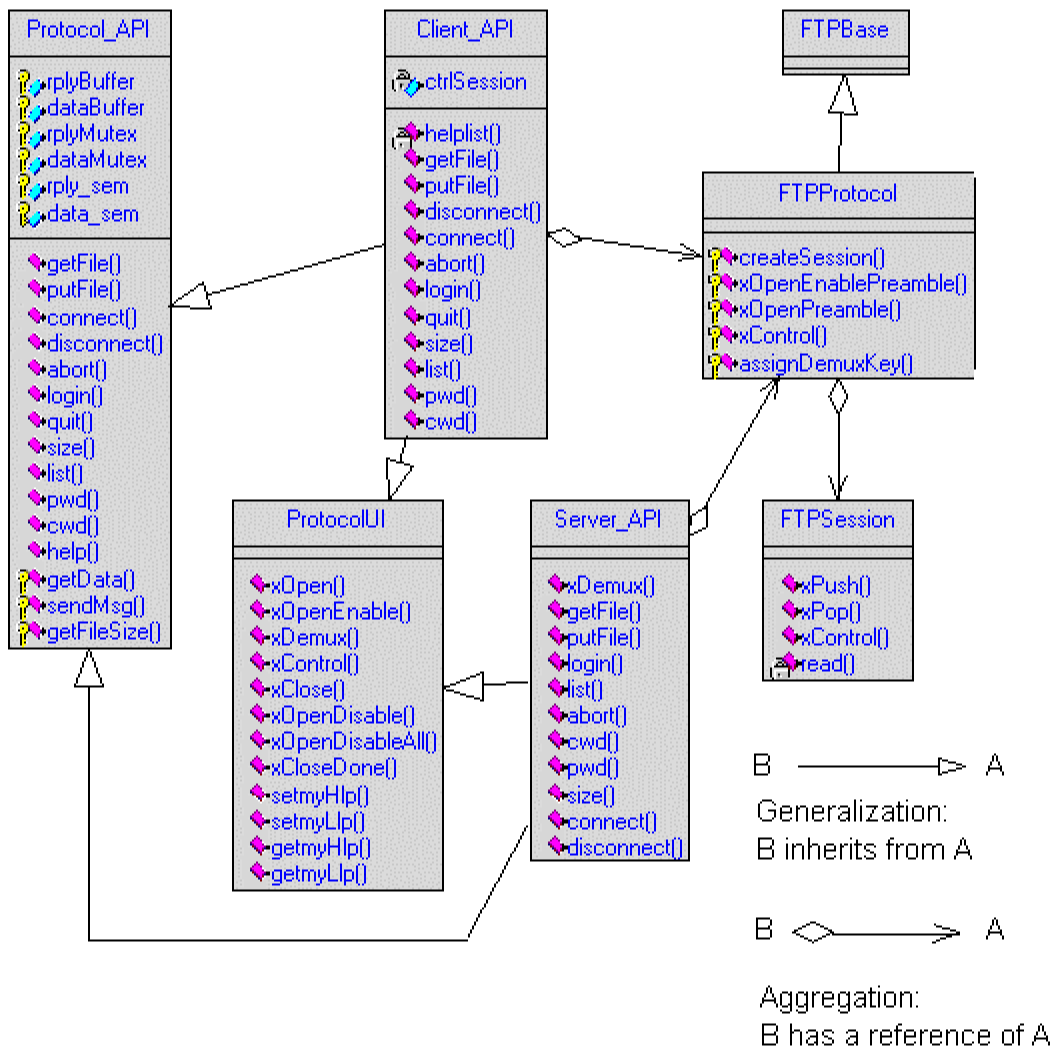

Figure 4.2 Class diagram.

The following is a description of each class displayed in this diagram.

- ♦ ProtocolUI

ProtocolUI is a class defined by PIX to provide a uniform protocol interface. Each method in this class is a virtual function and therefore can be overridden by the concrete derived protocol. The xOpen() method is used by a high-level protocol to actively open a session associated with its low-level protocol. The xOpenEnable() method is used by a high-level protocol to passively open a session associated with the low-level protocol. A passive open indicates a willingness to accept connections by the remote participant as discussed in Section 2.3.3. The xDemux() method is used by a low-level session to pass messages to the high-level protocol. The xControl() method is used by other protocols to retrieve information or set processing parameters. The xClose() method decrements the reference count of a session or, if the reference count is zero, deletes the session. The xOpenDisable() method is used by a high-level protocol to undo the effects of an earlier invocation of xOpenEnable. The xOpenDisableAll() method is used to cancel all xOpenEnable() operations. The xCloseDone() method is used by a low-level protocol to inform the high-level protocol that a peer participant has closed the session which was originally opened by the high-level protocol. The getmyHlp() and getmyLlp() methods are used to get a protocol's high-level and low-level protocols. The setmyHlp() and setmyLlp() methods are used to set a protocol's closely related high-level and low-level protocols, which are mainly used to build a protocol stack graph.

- ♦ FTPBase

FTPBase is produced by PIX's protocol generator, as shown in Figure 2.7 and conforms to UPI defined by ProtocolUI.

♦ FTPProtocol

FTPProtocol is inherited from FTPBase and is designed to support both the active sessions initiated by the client end and the passive sessions required by the server end. The protocol keeps separate collections of both active and passive sessions in active map and passive map. Each session is identified by a unique key. Unlike TCP or IP protocols, which embed keys in the protocol header, FTP's active key is its low-level session because there is a one-to-one map between an FTP session and a session of transport protocol. Passive sessions are defined by enable objects. An enable object contains the session's passive key and a count. FTP's passive key is unique because there is just one entity above the FTP protocol, namely the API layer. Whenever the API class calls the xOpenEnable() operation of FTP, the reference count of the enable object is incremented. The createSession() method creates an FTP session. The xOpenPreamble() method calls its low-level protocol's xOpen() to get the low-level session, and save this session as the active key. The xOpenEnablePreamble() method calls its low-level protocol's xOpenEnable(). The assignDemuxKey() method saves the session in the parameter list as the active key and is called by xClose() of the active open feature. The xControl() method gives FTP information requested by other protocols according to different operation codes.

♦ FTPSession

FTPSession represents an FTP channel between the client and server. When a message is to be sent out, the xPush() method of FTPSession is called, and when a message is received, the xPop() method is called. The receiving FTP session is responsible for reading its incoming data in chunks equal in size to the ones pushed at the sending side.

In other words, if it receives less than a chunk, it waits for more to come, and if it receives more than expected, it caches the surplus part until more data arrive to form another chunk. This algorithm is called buffer management and implemented in a private method read() and called by xPop() when data is received. Currently, the xControl() method provides the information on the total header length of its low-level protocols. Its capability could be extended by adding different operation codes to satisfy various requirements.

♦ Protocol_API

Protocol_API defines an interface for the FTP server and client applications. Its protected data members include the message buffers for control and data connections, the semaphores to synchronize multiple threads for different messages, and the mutex to ensure that only one thread is blocked. The public methods are the functional commands supported by our FTP implementation as listed in Section 4.2.1. They are pure virtual functions and the derived class gives the real definitions. The protected methods are helper functions to facilitate sending messages (sendMessage()), get file size (getFileSize()) and get data content from the message buffer (getData()) that are common to all derived classes.

♦ Server_API

Server_API implements two interfaces of different nature, one for the protocol stack, the other for FTP server functionality. Therefore, it inherits from both Protocol_API and ProtocolUI to provide service to server application. It decouples the server's services from its concrete implementation. It overrides the public methods in Protocol_API to define the server's response to user commands.

♦ Client_API

Similar to Server_API, Client_API also provides different interfaces for protocol stack and FTP client functionality respectively. It inherits from both Protocol_API and ProtocolUI and provides service to client applications. It hides the user from concrete service implementation. Its private data member *ctrlSession* is dedicated to the FTP control connection. It overrides every public method in Protocol_API to give the client's interpretation to user's commands.

4.2 Enhancement to Map Tool in PIX

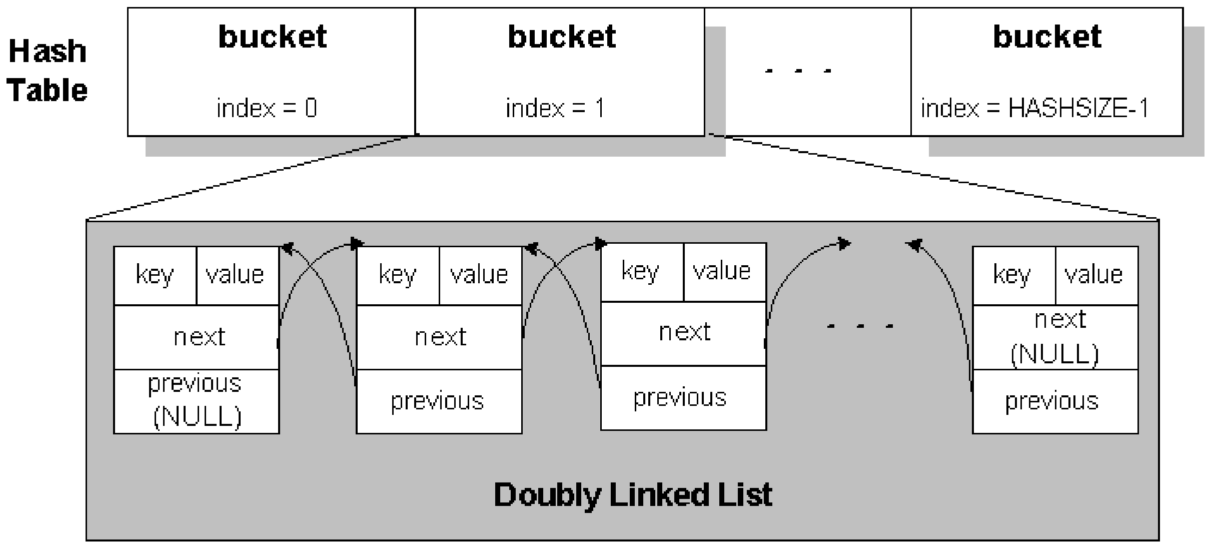

Figure 4.3 Data structures of map in PIX.

Our current PIX implementation uses a map tool to translate protocol specific identifiers of incoming packets into session objects. Each entry of the map is called a map element, which is a key-value pair called a binding. The design of map tool in our current PIX implementation consists of two interwoven data structures as pictured in Figure 4.3.

1. A hash table comprised of an array of buckets.
2. A doubly linked list of map elements.

- Hash Table

A hash table is a dictionary in which keys are mapped to array positions by a hash function. When doing a lookup operation, rather than searching through all the elements of the hash table for a matching key, a hashing function analyses a key and returns an index number. This index matches a stored value, and the data is then accessed. In our design, a division method is used as the hash function. It is a common choice when very little is known in advance about the keys. This hashing function works by adding each character of a key to a scrambled combination of the previous ones and returning the reminder modulo the array size. In the design, the size of the hash table (HASHSIZE) and the size of the key (KEYSIZE) are determined through the template parameters, which serve as a configuration repository to define a map.

- Hash Table Bucket

Each element in the hash table array is called a hash table bucket and is represented by a doubly linked list in our design. Every hash table bucket accommodates the map elements which contain the same hash value. Using a linked list is advantageous that there is no *priori* limit on the number of elements a map can hold, and at most one hash computation is needed for any map operation. Using a doubly linked list as opposed to a singly linked list provides extra flexibility on predecessor queries at the cost of doubling the number of pointers. As for computational complexity, both searching and deleting are O(n) operations in a singly linked list and in a doubly linked list [Dro99].

The map class supports following operations:

- *install*: determines whether the key being installed is present. If so, the new definition suppresses the old one, otherwise a new entry is created.
- *erase*: deletes an existing entry in the map.
- *lookup*: searches for an entry in the map and returns a pointer to it. If the entry is not present, returns NULL.
- *resolve*: finds an entry in the map and returns the value of this entry. If the entry is not present, it returns NULL. This is the function used by protocols from PIX to find the right session associated with an incoming message.
- b*egin*: finds the first non-empty entry in the hash table.
- *next*: starting from a given point, finds the next non-empty element in the map.

Typically each protocol that has a session will perform a lookup operation provided by the map tool for each incoming message to route it to the appropriate session. Thus, lookup is on the critical path of protocol processing and its performance is crucial. In the original implementation, lookup adopts the following procedure. Given a target element *e*, with key (*k*) and value (*v*), first calculate the index (*i*) of the bucket in which the element is possibly stored by the hash function. Then, search through the doubly linked list referred to by *i* to find *e*, by comparing *v* with value of each node in the list. Thus, the operation returns either a pointer to the node with matching value or, if not found, a NULL pointer.

From the observation and analysis of test results, we found that current map is not efficient enough in the general case of one session transferring bulk data back-to-back and it eventually becomes a performance bottleneck in our current PIX implementation. We solve this problem based on a mechanism defined in x-Kernel [Mos96^2]. Assuming locality of network traffic, given that a session *s* has received a packet, it is highly probable that *s* will also be the session receiving the following packet. Keeping this in mind, we can improve the efficiency of searching by caching the most recent updated or referred session. The proposed solution is as follows. Add a pointer to a node of the doubly liked list, called cache. After a certain operation to the map, for example, a successful lookup, the updated map element is stored in this one-entry cache. In the common case of one session receiving multiple packets back-to-back, this avoids a potential bucket-list traversal for all except the first lookup. Cache is updated in several cases:

1. After a successful installation of one map element to the hash table, the cache points to the element which was just added.
2. After a successful lookup, the map element that was found is stored in cache. The next lookup will check this cached element before doing a regular hash table lookup.
3. After a successful erase, cache is set to the NULL pointer.

This enhancement significantly increases the efficiency of the mapping operation. In the next chapter, we give performance results to demonstrate this enhancement.

4.3 FTP Implementation in x-Kernel

As discussed in Section 2.3.2 and Section 2.3.3, PIX borrows the fundamental ideas concerning protocol, session, message, map and event handler from x-Kernel. Therefore, the way the FTP implementation is designed in x-Kernel is very much similar to the design in PIX. The x-Kernel version, however, is written in structured programming language, *i.e.* C. The x-Kernel is object-based and provides its own object infrastructure, which becomes the glue to allow one object to invoke an operation on another object. For example, when an object invokes the operation on some other protocol, it includes this protocol as one of the arguments to this operation. The main functions defined by FTP in x-Kernel are listed in Figure 4.4. The UPI functions declarations are mandatory for each protocol in x-Kernel. The ftp_init() function is called at system startup time. It initializes the FTP protocol object, including protocol state and maps. The ftpOpen(), ftpOpenEnable(), ftpCreateSessn(), ftpDemux(), ftpControlProtl(), ftpClose() and getproc_protl() functions could be mapped to xOpen(), xOpenEnable(), createSession(), xDemux(), xControl(), xClose(), getmyHlp() and getmyLlp() methods in the FTPProtocol class of PIX. Similarly, ftpPush(), ftpPop(), ftpControlSessn() functions could be mapped to xPush(), xPop() and xControl() methods in the FTP session class of PIX. The FTP functions now only include the basic send and receive files capabilities just for testing purposes. Other functional commands are absent. Private helper functions work the same as those of FTP in PIX.

4.4 Differences between Implementing Protocols in PIX and x-Kernel

Besides the difference in programming languages employed by PIX and x-Kernel, there are other distinctions in implementing protocols in these two frameworks as well. The protocol generated from PIX is already able to accomplish basic protocol behavior, such as sending/receiving/demultiplexing messages. What developers need to do is to add protocol specific algorithms or rules to create a fully functional protocol. In contrast, in x-

```
// UPI function declarations
void                ftp_init (Protl);
static Sessn
       ftpOpen(Protl self, Protl hlp, Protl hlpType, Part *p);
static XkReturn
       ftpOpenEnable(Protl self, Protl hlp, Protl hlpType, Part *p);
static Sessn
ftpCreateSessn(Protl self, Protl hlp, Protl hlpType,
               ActiveId *key);
static XkReturn  ftpDemux(Protl self, Sessn lls, Msg *dg);
static XkHandle  ftpPush(Sessn self, Msg *msg);
static XkReturn  ftpPop(Sessn self, Sessn lls, Msg *msg, void *hdr);
static int
       ftpControlProtl(Protl self, int opcode, char *buf, int len);
static int
       ftpControlSessn(Sessn self, int opcode, char *buf, int len);
static Part *    ftpGetParticipants(Sessn self);
static XkReturn  ftpClose(Sessn s);

// internal function declarations
static void   getproc_protl(Protl p);
static void   getproc_sessn(Sessn s);

// FTP function declarations
XkReturn          clientGetFile(char * fname, Sessn lls );
XkReturn          clientPutFile(char * fname, Sessn lls );
XkReturn          serverGetFile(char * fname, Sessn lls );
XkReturn         serverPutFile(char * fname, Sessn lls );

// private helper functions
void          getData(char * buff, char * content);
int           getFileSize(char *name);
XkReturn          sendMsg( int type, char * content, Sessn llSession);
```

Figure 4.4 FTP functions prototypes in x-Kernel.

Kernel, protocol developers need to develop each component of a protocol, in conformance with UPI, to ensure interoperability with other layers. More specifically, each protocol from x-Kernel needs to implement its own active open, passive open, close and control connections, as well as demultiplex, send and receive messages functions. However, a protocol generated from PIX does not need to implement any of those operations unless it requires special processing; for example, IPv4 protocol with DSR ability in PIX needs to implement its own demultiplexing method (xDemux) to differentiate between characteristics of the message and achieve dynamic source routing. An FTP session needs to implement its own read functions (as part of xPop) to make sure that it reads incoming data in chunks equal in size to what has been pushed at the sending side.

4.5 Adapter Protocols

In order to give a fair comparison of FTP performance from PIX and x-Kernel, we need ensure that, the underlying transmission mechanisms these two FTP implementations based on are the same. For this purpose, we adopted standard TCP/IP implementations encapsulated by socket interface in Linux to realize network communications.

Neither PIX nor x-Kernel supports direct invocation of socket functions. An adapter protocol is developed in each of the framework to map the UPI methods to socket functions. For example xOpen() in PIX (or Open() in x-Kernel) is mapped to create a socket, and then connect it to the remote participant; xOpenEnable() in PIX (or

OpenEnable in x-Kernel) is mapped to create a socket, bind a local address and set the socket to a listening mode through a listen function. The change of transportation mechanism underneath should be transparent to the application protocol. With the adapter, whenever an application wants to send out a message, it pushes the message to its lower level protocol in which, the adapter protocol sends the message out through corresponding socket operation (send). The adapter protocol has a thread dedicated to read data from socket's receiving buffer and pop up the data to its upper-level protocol for processing.

Chapter 5

Performance over Wired Communication

This chapter reports on two sets of experiments conducted to evaluate the performance of PIX. The first one demonstrates the benefit obtained by adding cache capacity to Map utility of PIX. The second one aims to quantify the impact that PIX and generative programming have on protocol performance. The equipment used in the experiment is a pair of Intel Pentium III, 547 MHz and 192 MB PCs, with Mandrake 8.0 distribution of Linux. Each experiment is executed enough times to get a statistically significant average.

5.1 Map Library Experiments

The purpose of this experiment is to verify whether adding cache capability to the Map utility would result in noticeable performance improvement in the case of the network host receiving data back-to-back. The performance of the enhanced map has been measured and compared to an older version of the tool that did not support cache capability. The following table (Table 5.1) gives the description of the labels used in the description of test cases or figures plotted from these tests.

Label	Description
New map – cache hits	Cache hit means the value stored in cache is the same as the value of the target element.
New map – cache misses	Cache miss means the value stored in cache is different from the value of the target element.
Old map	Map manager without cache capacity.

Table 5.1 Description of labels used in map experiments.

Two scenarios are considered and measured, similar to what had been done to evaluate the map tool in x-Kernel [Mos96^2].

1. The time required for a lookup (map.resolve()) operation in maps created with different key sizes and numbers of bindings, where there is a cache hit and cache miss respectively.
2. The time saved when continuously receiving data from the same remote network host.

Keeping these goals in mind, we designed two groups of tests as described in subsection 5.1.1 and 5.1.2.

5.1.1 Lookup Process

To avoid the possible impact of key sizes, we measured the lookup time in maps created with keys of different sizes. We installed 100 to 1000 bindings to each map and tried to record the time needed to perform a lookup operation under two situations: cache hits and cache misses. The key of each binding is different enough to guarantee that hash values are not the same to avoid concentrating bindings in a few hash buckets. The results of lookup time are shown in Table 5.1. We observe that the size of the key does not impose a significant impact on the look up time. When cache hits, the lookup time is almost the same, however under the case of cache misses or using the old map, the lookup time increases with the number of bindings.

BINDINGS		100	200	300	400	500	600	700	800	900	1000
4 Bytes	Cache hits	16.8	16.8	16.7	16.9	16.7	16.8	17	17	16.9	17
	Cache misses	19.1	20.1	22.7	25.1	27.3	30	30.9	33.8	37	39.3
	Old map	19.1	20.1	22.3	24.8	25.9	29.9	30.7	33.7	37	39.2
7 Bytes	Cache hits	17	17.1	17	16.9	16.9	16.8	16.9	17	17	17
	Cache misses	19	20.2	22.4	24.7	26.8	30	31.8	34.1	37.4	40.8
	Old map	18.9	20	22.1	23.7	26	28.5	31.2	33.7	36.9	40
8 Bytes	Cache hits	17	17	17.1	17	17	17	17.1	17	17	17
	Cache misses	19	20.3	22.9	25.2	27.6	30.1	31.9	34.2	37.5	41
	Old map	19	20	22	23.9	26	28.7	31.3	33.8	37	40
12 Bytes	Cache hits	17.1	17.2	17.1	17.1	17.2	17.1	17.2	17.1	17	17
	Cache misses	19.2	20.5	22.9	25.5	27.8	30.3	31.9	34.4	37.5	41.2
	Old map	19.1	20.3	22.1	24	26.1	28.9	31.4	33.9	37.1	40.1
14 Bytes	Cache hit	17	17	17.3	17.2	17	17.2	17	17.1	17	17.2
	Cache misses	19.4	20.9	23.1	25.8	27.9	30.5	32.3	34.7	37.9	41.8
	Old map	19.2	20.5	22.3	24.1	26.3	29	31.6	33.9	37.1	40.5

Table 5.2 Lookup time, in microseconds, in maps of different sizes of key.

To get a clearer view of comparison, the results of doing lookups in a map with key size of eight bytes are plotted as Figure 5.1. The time to do a lookup when there is a cache hit could be approximated to a constant of 17 microseconds, independent of the number of bindings. In contrast, when cache misses, the lookup time increases, as the number of bindings increases, with a tendency of around ten percent. We also observed that the curve corresponds to the lookup time of new map, and when cache missing almost overlaps the curve represents looking up in the old map. This proves that adding a cache mechanism to map does not add much overhead to lookup.

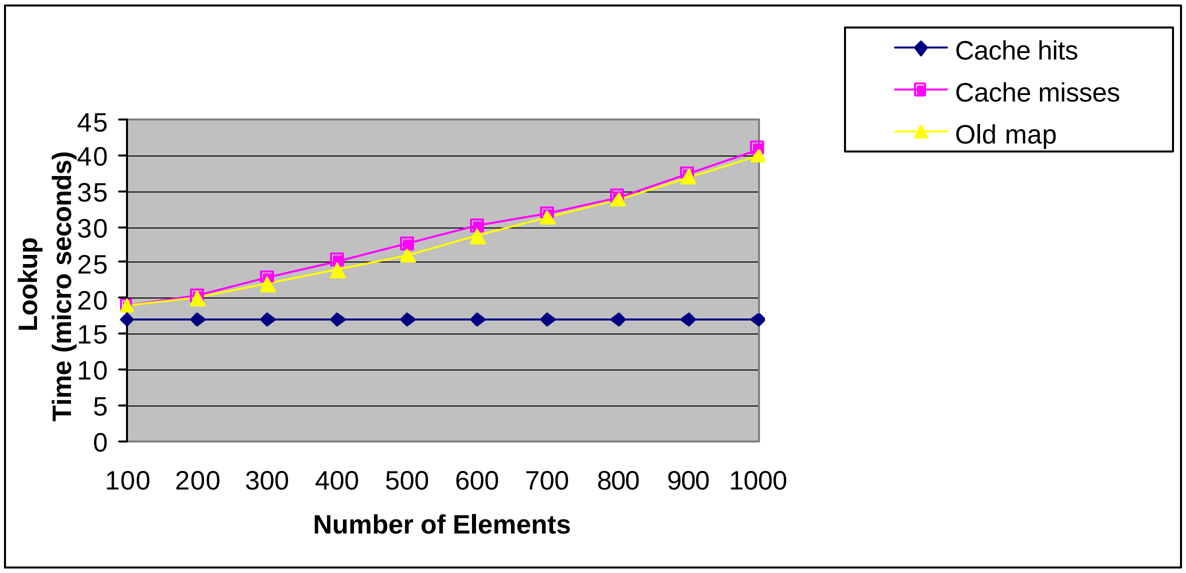

Figure 5.1 Lookup time in a map of key size of eight bytes.

5.1.2 xDemux Process

The experiments described in the previous subsection address the more general objective of improving the lookup process by using a cache mechanism. This subsection will focus on a specific case: how does the enhancement expedite data processing in a protocol? The xDemux method of a protocol is the one that uses the lookup procedure. It maps identifiers in an incoming message to appropriate sessions in the map maintained by a protocol. Therefore we try to demonstrate that the enhanced map is more efficient in the case that one session receives multiple packets back-to-back.

The aforementioned apparatus are connected by a null modem. Simple clients and server application run on separate machines, communicating through a protocol stack (discussed in detail in Chapter 6) wholly developed from PIX, which supports the xDemux operation. Each experiment involves transferring different sizes of messages 50 times. The xDemux time is measured from the time a message arrives at the physical layer of the server until right after the message is processed in the network layer.

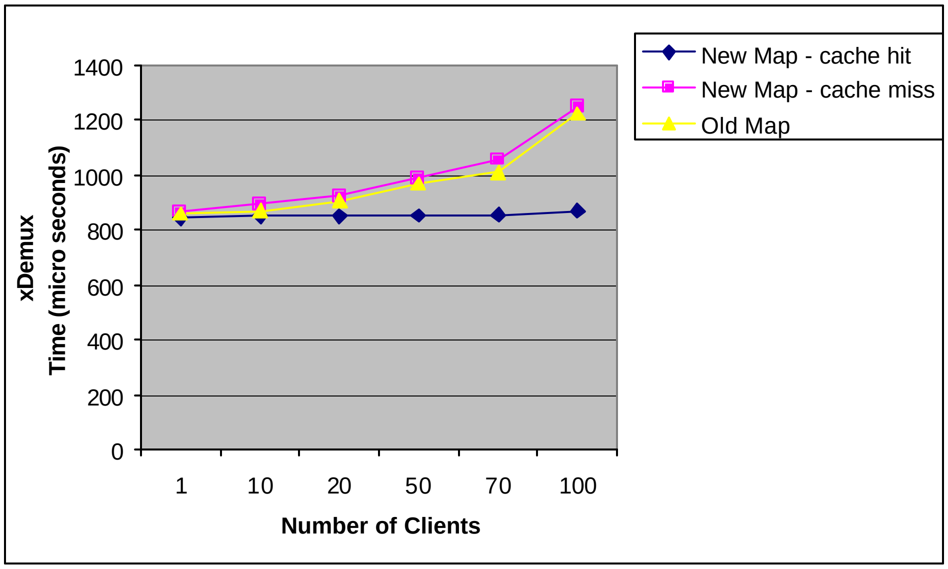

Figure 5.2 Time of xDemux as the function of number of clients.

When one server deals with multiple clients, the number of clients is the variable that determines the number of bindings in server protocols' maps. Therefore, it is an important factor affects the speed of xDemux. Figure 5.2 shows the operation time of xDemux as a function of number of clients when transferring a one byte message to the server. A phenomenon similar to that seen in Figure 5.1 is observed: when cache hits, the xDemux time is kept approximately constant, while the performance of cache missing and the old map are almost the same.

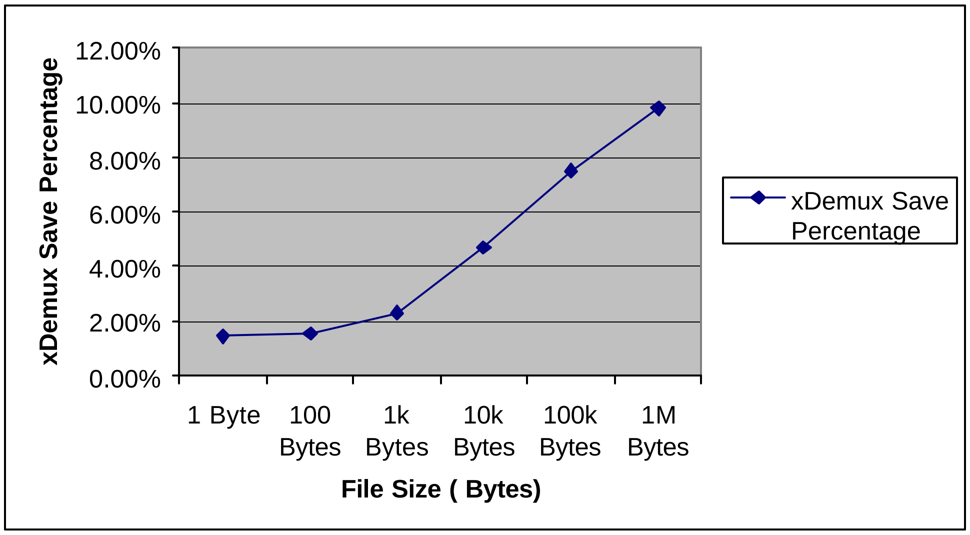

Figure 5.3 xDemux improvements with cache.

The size of messages determines the times of lookups which will be successively conducted for a certain network circuit. When one network host receives bulk data from another network host, the possibility of having cache hit is much higher. Therefore, we address this situation by transferring different size of messages between one server and one client. When message size is greater than one packet, it is divided into several packets. Accordingly, the receiver side performs multiple xDemux operations for one whole message, and the overall xDemux time is recorded as the sum of all these separate xDemux operations. The curve in Figure5.3 represents the time saved by xDemux using new Map with cache, as compared with using the old one without cache, in percentage terms. We could see that adding cache capability to map library improves performance.

When the file is small, the improvement is not very evident; however, as the file size increases, it becomes more apparent. This proves our assumption that cache will improve performance in the case of receiving continuous data from the same network host.

5.2 Protocol Performance Analysis

This set of experiments compares the performance of a file transfer protocol implemented in PIX with an x-Kernel implementation and a Linux implementations (NcFTP). Firstly, we measure the latency and throughput of the file transfer protocol implementations. Secondly, CPU and memory usage are measured.

5.2.1 Experimental Environment

The same Linux machines as used in Section 5.1.2 are used. They are connected by a 100-Mbps Ethernet crossover cable. The implementations are evaluated using the Mandrake 8.0 distribution of Linux. For the PIX experiment, each machine is configured with a version of the software embedding the FTP-Adapter protocol stack, as illustrated in Figure 5.4. One machine hosts the client side while the other hosts the server side. The Adapter protocol converts the API of PIX, to the API of socket. For the Linux experiment, the NcFTP implementation is used. The x-Kernel implementation is interfaced with the socket API, also using a similar Adapter protocol.

The experiment consists of running the same test 50 times. Each test consists of retrieving a file on a server by a client. The system clock is read at the beginning and end of each test. The average of the 50 runs is reported. One thing worth noting is that NcFTP reports to the user the duration of the time interval starting immediately after a data connection has been opened successfully and ending immediately after all the data has been transferred and the data connection is fully closed. Indeed, in general, a user does not care about the setup time associated with establishing connections and exchanging control information. Taking all the overhead into account is important in our experiments because we are concerned about the impact of the whole protocol stack on performance. Therefore, we consistently define the time it takes to retrieve a file in PIX, NcFTP and x-Kernel as the period of time beginning when a client starts to process a retrieval request and ending when the processing of the request is finished.

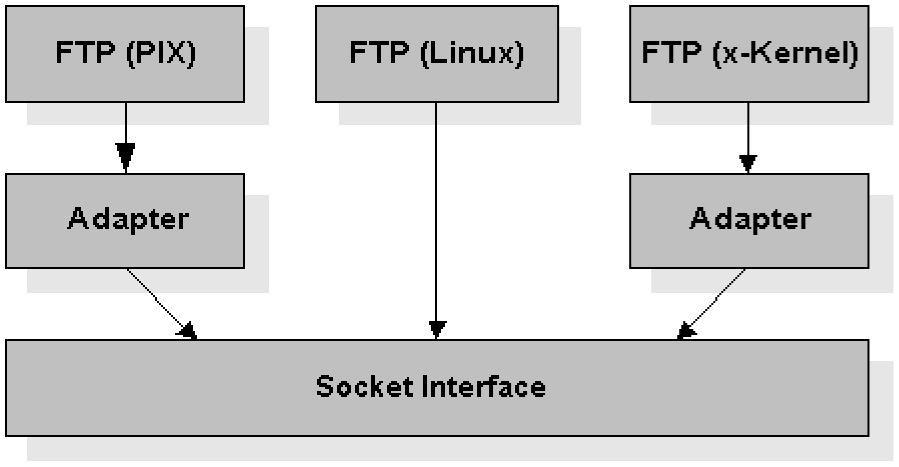

Figure 5.4 Protocol graph of PIX implementation, Linux implementation and x-Kernel implementation.

5.2.2 Latency Result

Table 5.3 gives the results for transferring files with sizes of 1B to 100MB, and Figure 5.5 plots the results of Table 5.3 in a graph. The horizontal axis is log_{10} of file size in bytes while vertical axis is log_{10} of time in milliseconds. Figure 5.6 zooms in files under 100 KB. The PIX stack takes slightly more time than the x-Kernel stack, which takes slightly more time than NcFTP stack. The differences, however, tend to be not noticeable as the size of files grows.

File Size	1 Byte	100 Bytes	1k Bytes	10k Bytes	100k Bytes	1M Bytes	10M Bytes	100M Bytes
PIX	0.0353	0.0358	0.0375	0.0413	0.047	0.118	0.965	10.45
x-Kernel	0.0348	0.0353	0.037	0.0407	0.0454	0.115	0.937	10.17
NcFTP	0.0341	0.0342	0.0361	0.0391	0.0436	0.11	0.9	9.75

Table 5.3 Latency in seconds, of file transfer in PIX, x-Kernel and NcFTP.

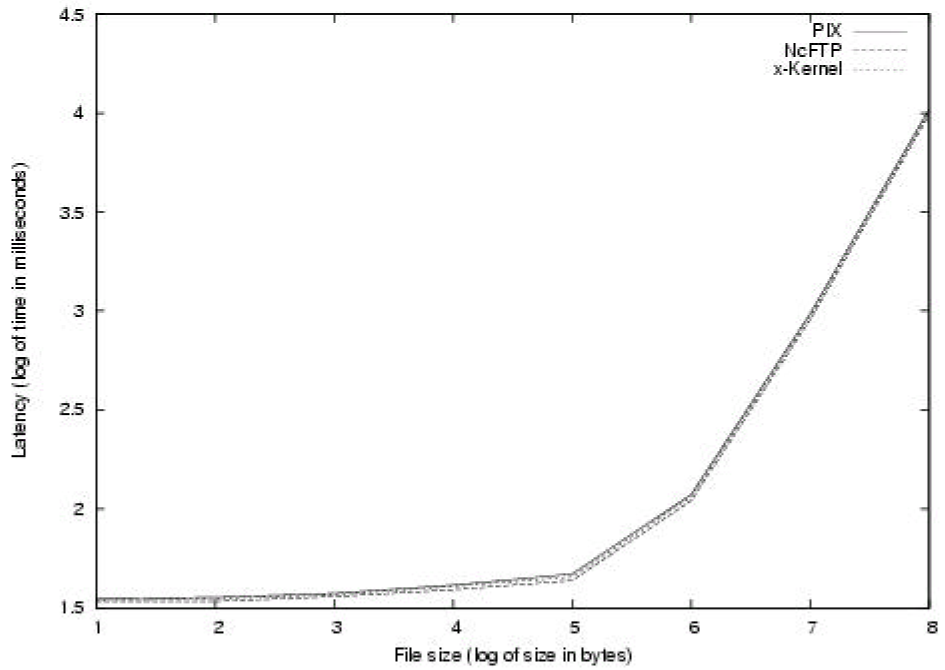

Figure 5.5 Latency of file transfer in PIX, NcFTP and x-Kernel (log_{10}-log_{10} scale).

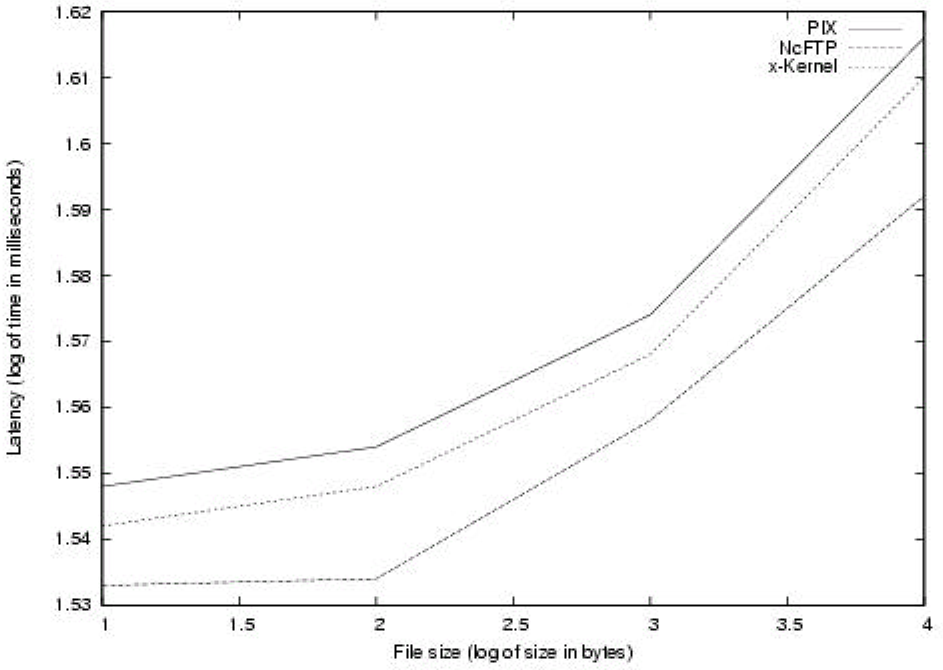

Figure 5.6 Latency of transfer of files under 100 KB (*log_{10}*-*log_{10}* scale).

To analyze further the extent to which the PIX stack is slower than the NcFTP implementation, we calculate the proportion of increment in latency, denoted as *p* and defined as follows:

$$p = \frac{T_{PIX} - T_{NcFTP}}{T_{NcFTP}}$$

where T_{NcFTP} is the latency of NcFTP and T_{PIX} is the latency of PIX. The value of *p* oscillates between 3.5%, for the smallest file, and 7.2%, for the largest file. If we replace the latency of NcFTP T_{NcFTP} in the equation with the latency of x-Kernel $T_{x\text{-}Kernel}$, then the value of p oscillates between 1.5% (smallest file) and 2.8% (largest file). This is a measure of the extent to which PIX is slower than NcFTP and x-Kernel.

5.2.3 Throughput Result

The throughput, in M bps, is defined as follows:

$$\frac{8S}{1024^2 L}$$

where S is the file size in bytes and L is the latency in seconds. The throughput for all three implementations, as a function of file size, is given in Table 5.4. All three implementations are capable of saturating the 100 Mbps network, and 100% efficiency is not possible because of the overhead introduced by the protocol layers.

File Size	1k Bytes	10k Bytes	100k Bytes	1M Bytes	10M Bytes	100M Bytes
PIX	0.208	1.89	16.62	67.80	82.90	76.55
x-Kernel	0.211	1.92	17.21	69.57	85.38	78.66
NcFTP	0.216	2.00	17.92	72.73	88.89	82.05

Table 5.4 Throughput in M bps, of file transfer in PIX, x-Kernel and NcFTP.

The throughput of transferring files of size 100 KB and under is low because the network pipe cannot be fully utilized. The table also shows that transferring files of size 10 MB achieves higher throughput than transferring files of size 100 MB. One possible reason is that transferring 100 MB files involves a huge number of file system accesses, which reduce the system throughput.

5.2.4 Resource Usage

Resource consumption is another important performance criterion. Usually it refers to the CPU and memory usage for executing a program.

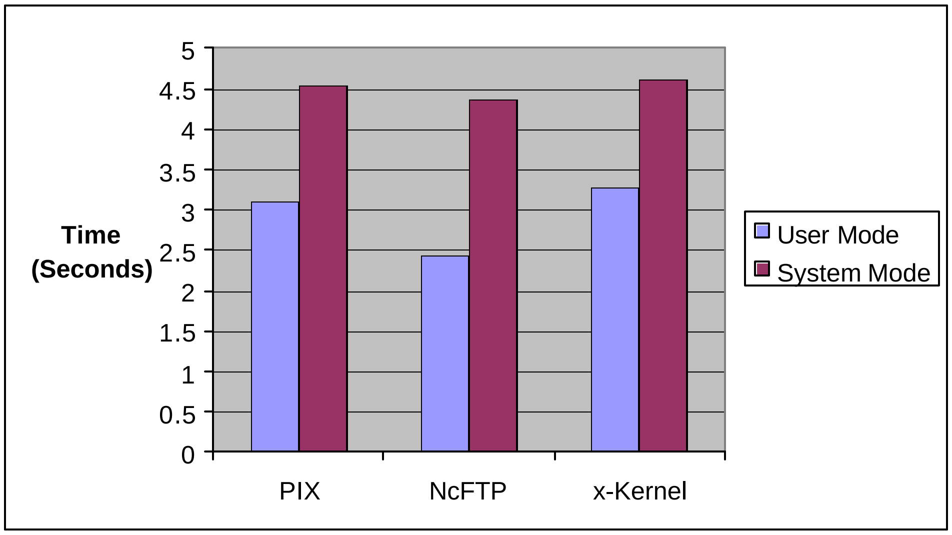

Fig 5.7 CPU time usage.

We use the *getrusage* system call of Linux to collect information about the CPU time spent by an FTP client executing in user mode or in system mode during the process of retrieving a file from a server. The transfer of large files is evaluated, as they demand higher CPU time. The usage of the CPU is measured during the execution of a file retrieval procedure. This experiment is repeated 50 times and the CPU time usage is averaged. Figure 5.7 represents the CPU time in the user mode and the CPU time in the system mode when a client retrieves a 100 MB file. We observe that the CPU system time used by the three implementations is very close, while the PIX protocol stack and x-Kernel protocol stack require noticeably more user time than NcFTP.

To determine the memory footprint of a process in real-time, we use the *top* command of Linux which returns the dynamic information regarding the memory usage of a specific process. Table 5.5 shows the memory footprint of FTP servers and FTP clients in KB. The column under the field of *Initialization* lists how much memory is used after the FTP server or client has been initialized and is ready to process a request. During the procedure of transferring a file, the memory usage increases, and then drops back to the initial state when the transfer is finished. The increase is related to the size of files being transferred: the larger the file, the bigger the increase. The column under the field of *100 MB File Transfer* shows the peak value of memory usage, which occurs during the transfer a 100 MB file. The NcFTP software occupies more memory because it supports complete FTP functionalities, while the FTP of PIX implements part of them and the FTP of x-Kernel only supports the file transfer capability.

Phase	Initialization Server/Client	100 MB File Transfer Server/Client
PIX	768/848	804/904
x-Kernel	672/664	688/700
NcFTP	1288/1152	1288/1224

Table 5.5 Footprint, in KB, of file transfer in PIX, NcFTP and x-Kernel.

Chapter 6

Performance over Packet Radio

Packet radio transmission refers to the breaking up of large blocks of data into small packets, called units, and sending them using radio signals. It is of special interest because of its unique characteristics and wide deployment in wireless communications. The characteristics of packet radio are multi hop networks, wide range, slow speed, short frames and high latency. There are several well-known projects which deploy packet radio. General Packet Radio Service (GPRS) is a bearer service of Global System for Mobile (GSM) phase 2+ system to access packet data networks. It applies a packet radio principle to transfer data packets between GSM mobile stations and external packet data networks. The data transfer rates could be up to several ten Kbps [CG97]. Mobitex [KK95] is also a packet-switched communication system. It provides seamless roaming and reliable transmission with a proven reliability factor greater than 99.99 percent. It supports a data rate of 8 kbps. Cellular digital packet data (CDPD) transmits packet data over idle cellular channels with a raw data rate of 19.2 kbps [CDP95]. The radius of a CDPD is typically limited to less than 10 miles. The point-to-point UHF 9.6 Kbps data radio was used for the Lander to Rover communications link of the Mars Pathfinder mission [MP96]. Although it was not packet radio *per se*, for example, there were neither encapsulation of data with headers nor addressing or routing, it had some of the transmission characteristics that we find in packet radio, such as a low bit rate. CubeSat is an example of space communications using packet radio [HPM$^+$00]. It is a cube with

10cm side and 1kg in mass equipped with data radios which operate over amateur radio frequencies. An example of CubeSat computer is as follows. Its on board computer (OBC) is a single board computer operating at up to 40 MHz with 128 KB erasable programmable read only memory (EPROM), 2 MB static random access memory (SRAM) and 32 MB of external flash memory. With the assumption of 97 minutes to orbit, with 60 minutes of sunlight, the power budget for the OBC is 0.61 W. It supports data rates of 1.2, 2.4, and 4.8 Kbps. On average, there are four contact periods a day with a total daily time of between 28 and 29 minutes.

In this chapter, we explore the performance of an FTP implementation that integrates with eXtended Satellite Transfer Protocol (XSTP), IPv4 with Dynamic Source Routing (DSR), AX.25 and Packet Assembler Dissembler (PAD) protocols, which are exclusively developed from PIX with GP model over packet radio. The experiments are conducted with data transfer rates of 1.2/9.6 Kbps, frame size 255 bytes, using medium access control protocol: Carrier Sense Multiple Access (CSMA) [AZ03]. We describe the protocol stack in details in Section 6.1, then give a brief overview of the architectures of related protocol stacks in space communications and, lastly, present the performance analysis.

6.1 Protocol Stack over Packet Radio

Currently a protocol stack as shown in Figure 6.1, that performs over packet radio transmission is under development in our Lab. The protocol stack aims to serve as the communication software for small satellites (spacecraft of in-orbit mass less than 500kg), initially a CubeSat. A file transfer protocol provides the capabilities of transferring files between a base station and a small satellite. It will be operated on XSTP [Ela03][EZB+03], which is an adapted TCP protocol specially designed for dealing with issues in Low Earth Orbit (LEO) satellite communication, such as intermittent connectivity and high bit-error rates. A subset of IPv4 with DSR [JMH+01] serves in the network layer. An AX.25 protocol [BNJ97] in Keep It Simple Stupid (KISS) [CK97] mode for packet radio is used in the link layer. The channel access of AX.25 is handled by CSMA. A PAD protocol acts as the device driver to read from and write to the hardware. The design and implementation of this whole protocol stack from scratch demands significant efforts, though with the aid of PIX the work turns to be much easier. The prototypes of each layer of protocol could be generated from PIX with appropriate specifications, and then the necessary protocol algorithms can be added to the prototype to get fully functional protocols.

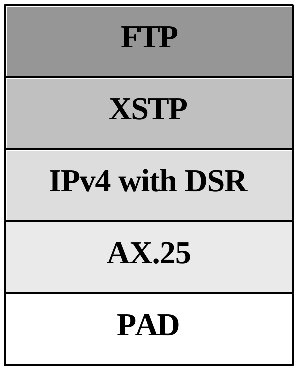

Figure 6.1 Software protocol stack for packet radio.

The FTP as part of the protocol stack discussed here is based on the design described in Section 4.1, though minor revisions have been made as discussed in Section 6.3.

6.2 Related Protocol Stack Architectures

In this section, we review the architectures of two protocol stacks of space communications realizing file transfer capabilities. One is Packet Radio Satellite (PACSAT) protocol suite used for LEO satellite communications, and the other is Space Communications Protocol Specification - File Protocol (SCPS-FP) defined by the Consultative Committee for Space Data Systems (CCSDS). The Cosmic Hot Interstellar Plasma Spectrometer Satellite (CHIPSat), as a newly launched micro-satellite that emphasizes in its official announcement the successful completion of FTP tests, is also briefly described.

6.2.1 PACSAT Protocol Suite

PACSAT is a generic term in the radio amateur service for a LEO satellite that carries on-board memory for the purpose of data storage and retrieval by ground stations [PW90[1]]. PACSAT is a server and the ground station is a client. The PACSAT protocol suite was developed at the University of Surrey to realize the potential of moving data between PACSATs and ground stations. The suite is composed of two protocols as illustrated in Figure 6.2: a File Transfer Level 0 (FTL0) protocol and a PACSAT Broadcast Protocol (PBP) that both operate above the AX.25 protocol.

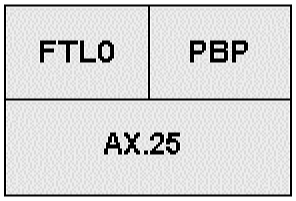

Figure 6.2 PACSAT protocol suite.

FTL0 [WP90] provides procedures for transferring binary files to and from a PACSAT bird using an error-corrected AX.25 connection. It supports the basic file transfer functions as well as file selection and directory capabilities with a command line user interface. FTL0 also supports simultaneous bi-directional file transfer and the resuming interrupted file transfer feature. PBP [PW90^2] takes advantage of the broadcast nature that all ground terminals in the satellite footprint could receive the satellite downlink signal. Thus PBP sends a single copy of common interest files once which could be received by all possible users. This significantly reduces the loads on both the satellite and the terrestrial network. PBP uses the AX.25 connectionless mode via Unnumbered Information (UI) frames.

The purpose of PACSAT is to store and forward data. This simple goal has been achieved by using AX.25 frames directly, which obstructs interoperability with TCP/IP networks.

6.2.2 SCPS-FP

The Space Communications Protocol Standards (SCPS) [SCP98] project aims at providing a suite of standard data handling protocols to facilitate communications with a remote space vehicle. The SCPS protocol stack includes a File Protocol (FP) as an application layer to exchange information, a Transport Protocol (TP) to address transport requirements in space, a Security Protocol (SP) that provides a protection mechanism to ensure integrity and security of message exchange, and a Network Protocol (NP) that supports both connectionless and connection-oriented routing of messages through networks containing space or other wireless data links, as shown in Figure 6.3. SCPS-FP provides end-to-end application layer services for flight missions in the space environment, including the uploading of spacecraft commands and software, and the downloading of collections of telemetry data. It is based on the Internet FTP and is generally interoperable with the latter. However, the project proposes extensions to meet the unique requirements of the space flight operations. The enhancements could be summarized as follows: 1) automatic restart of a failed transfer; 2) interrupt file transfer to enable resuming from a certain interrupted point; 3) ensure file and record integrity in case of abnormal connection loss; and 4) suppress reply text to achieve bandwidth efficiency.

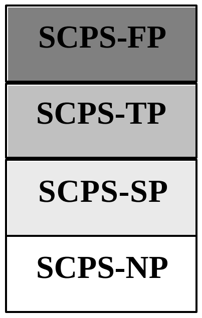

Figure 6.3 SCPS protocol stack.

The SCPS protocol suite is resource-intensive and generally only installed on large-scale spacecrafts rather than on small satellites.

6.2.3 CHIPSat

CHIPSat [DWS02] is the first University Explorer (UNEX) mission funded by National Aeronautics & Space Administration (NASA). It was launched on January 12, 2003 from Vandenberg Air Force Base. It is the first US mission that uses end-to-end satellite operations with TCP/IP and FTP. The idea of end-to-end operation on spacecraft over TCP/IP internet protocol suite has been analyzed and demonstrated by NASA OMNI team via UoSat-12; however, CHIPSat is the first attempt to implement this concept as the primary means of satellite communications. The TCP/IP and FTP protocol stack is built into the WindRiver VxWorks Real Time Operating System (RTOS) in the spacecraft single board computer to move data between spacecraft and the ground system. There are no details regarding the FTP implementation and testing of CHIPSat available at the time this thesis is being written.

6.3 Performance Analysis

6.3.1 Experimental Environment

The same two PCs as described in Chapter 5 are used for our experiment here, one configured as a server (to simulate a satellite) and the other as a client (to simulate a ground station). Each of them is connected to a Kenwood TM-D700A/E FM transceiver to transmit and receive packets. This kind of transceiver contains a built-in terminal node controller (TNC) that converts packets to audio tones and vice versa. The TNC conforms to the AX.25 protocol and implements framing and CSMA. The frequency selected is 432.000 MHz, which is relatively uncongested compared to other amateur frequencies. The transceivers are put into KISS mode to enable data transfer in frames. The PAD frame size is 255 bytes, which is the maximum length of the data portion of a packet supported by our radio. This transceiver provides two data transfer rate 1.2 or 9.6 Kbps, and we performed tests using both of them. Tests were performed in a naturally noisy environment. Each test involves retrieving a file of certain size from the server to the client. Each test is performed 20-30 times, and results which have been dramatically affected by noise are discarded, while the average of reasonable data is reported.

Compared with wired communications, radio transmission is slow. It is restricted by the processing speed and frequency between the computer interface and the device. From preliminary testing, we found that it takes around one second to finish TCP three-way handshake to get connected with the data rate of 9.6 Kbps. To expedite the file transfer procedure, we modified the FTP designed in Section 4.1 by merging the FTP control and data connections. This is doable because while files are transferred over a data connection, most of the time the control connection is idle. Accordingly, we define latency as comprising the time required to send requests, replies and file data, excluding the time to open and close a separate data connection.

6.3.2 Result Analysis

Figure 6.4 graphs the results of transferring files with different sizes from 1B to 10KB with a data transferring rate of 9.6 Kbps and 1.2 Kbps respectively. The horizontal axis is log_{10} of file size in bytes while the vertical axis is log_{10} of time in seconds. We observe that when transferring files of small size (under 1 KB), bandwidth is the primary determinant of the latency, as files done in 9.6 Kbps are much faster than using 1.2 Kbps. As file size increases, it involves many more incidents of writing to the TNC device, which slows down the whole transfer process. To illustrate this, we calculate how much FTP data could be contained in each frame as follows: the maximum PAD frame size is 255 bytes, an AX.25 header is 17 bytes, an IP header is 60 bytes (including source routing options), an XSTP header is 16 bytes, and an FTP header is 5 bytes. Therefore, each frame could only carry at most 157 bytes of FTP data. In turn, a 10 KB file needs to be fragmented into and transmitted in 66 frames. In this case, the processing time is dominant over transmission time because the high number of small packets.

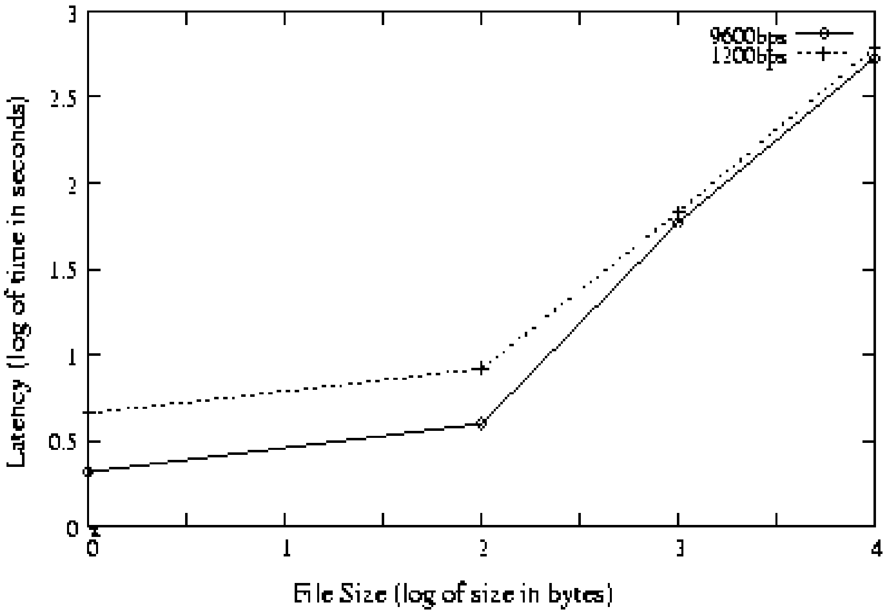

Figure 6.4 Latency of transfer of files under 10 KB (log_{10}-log_{10} scale).

The throughput results, in bps, of transferring files under 10 KB are organized into Table 6.1. It is interesting to note that better throughput is achieved when transferring files of small size, which is contrary to what occurs when files are transferred in wired communication. This is due to the high processing time of large files, as illustrated above. We express throughput as a fraction of the available bit rate, referred to as normalized throughput, and the results are shown in Table 6.2. The average normalized throughput achieved in 1.2 Kbps is around 0.24, while with 9.6 Kbps, the normalized throughput is pretty low. The AX.25 uses *p*-Persistent CSMA scheme for medium access control (MAC), where *p* is the probability of the transmission when medium is idle. According to analytic models [Klei75], the global throughput can theoretically be pushed very high by increasing the global offered channel traffic. We believe that, a normalized CSMA throughput of 0.24 observed locally by a client under real conditions and light load (two nodes only: one simulates ground station and the other simulates satellite) is consistent with theoretical estimations.

File Size	1 Byte	100 Bytes	1k Bytes	10K Bytes
1.2 Kbps	334.07	375.21	212.38	220.26
9.6 Kbps	758.86	768.16	240.92	248.42

Table 6.1 Throughput of transfer of files under 10 KB in bps.

File Size	1 Byte	100 Bytes	1k Bytes	10K Bytes
1.2 Kbps	0.28	0.31	0.18	0.18
9.6 Kbps	0.08	0.08	0.03	0.03

Table 6.2 Normalized throughput of transfer of files under 10 KB.

Chapter 7

Conclusion and Future Work

7.1 Concluding Remarks

Generative Programming has attracted attention due to its claimed benefits of modeling families of systems and automatic generation of highly customized and optimized system according to a particular specification. PIX is an example of applying the GP paradigm in the network communications domain. In a companion paper [BB02], it is argued that, with respect to more traditional protocol development tools, PIX offers configurability superiority because of the use of generative programming. An important issue that needed to be clarified, was whether or not this configurability superiority was at the expense of performance. By means of the file transfer protocol case, this thesis has analyzed the performance of PIX and GP based protocol implementation with respect to traditional approaches. Furthermore, the performance of a stack of protocols exclusively developed from PIX over packet radio transmission has been presented as a case study.

The work of this thesis can be examined from two aspects. The first aspect focuses on the general performance of PIX and GP based protocol implementation, which is comprised the following tasks:

1) Developing a file transfer protocol from PIX and x-Kernel frameworks respectively.

2) Developing an Adapter protocol from PIX and x-Kernel framework to make a fair testbed to conduct performance experiments.

3) Analyzing performance in the latency, throughput and computing resources usage of PIX and GP based, x-Kernel based and Linux socket based FTP implementations.

Regarding latency, we found that PIX and GP are 3.5% to 7.2% slower than NcFTP (structured programming) and 1.5% to 2.8% slower than x-Kernel (structured and object-based programming). Concerning throughput PIX can achieve 92% of NcFTP's peak throughput and 97% of x-Kernel's peak throughput (the 10 MB column in Table 5.4). Relating to resource usage, CPU usage time is of the same order with all three implementations. Memory usage by NcFTP is higher, but currently it supports more functionality.

Potentially, a system generated using the GP approach has more layers of inheritance than a system that would be coded by a programmer because in the GP approach each grammar rule is mapped to an inheritance relation. More layers of inheritance and the dynamic binding associated with it could be the main impact of using GP on performance of a system.

The second aspect of the research relates to our revision of the FTP developed in the first part of the work, and its configuration into the protocol stack entirely developed from PIX. The protocol stack runs over packet radio communications. Due to the small frame size permitted in packet radio, the latency is high compared with wired transmission. While the normalized peak throughput is comparable to the theoretical estimations.

7.2 Future Work

The work of this thesis could be extended in the following aspects: firstly, by enhancing current PIX framework, secondly, by completing the current FTP implementation to better accommodate to satellite communication, and lastly by improving performance over packet radio through header compression.

Presently, PIX lacks a buffer management mechanism between its transport layer protocol, i.e. XSTP and applications that based on it. The transport layer protocol controls the network traffic flow, therefore it caches the data to be sent from an application to a buffer and sends them out after a certain time interval. This effectively isolates the transport layer from the application's sending behavior. On the receiving side, the transport layer pops incoming segments to its upper layer in sequence. However the size of the messages passed up to the application may not correspond to those pushed down from the sending side, due to the possibility of segment loss and reordering in the network. The receiving application is responsible for reading incoming data in chunks equal in size to the ones pushed in sending application. This kind of mechanism is known as buffer management in socket interface and is implemented in our FTP application. This solution could be extended to PIX as a general feature to assist any application protocol that is facing the same problem.

The current FTP application that runs over packet radio could be enhanced to mitigate the challenges that occur in packet radio communications. Two features under consideration are the resumption of an interrupted file transfer from a certain point and the suppression of reply text to achieve better bandwidth efficiency.

From the analysis of our experiments conducted over packet radio, we know that the heavy overhead, introduced by each protocol layer, occupies a big portion of the frame, which impedes the performance. So an option to be considered to improve the performance is compression of headers.

Bibliography

[Ab93] M. B. Abbot, A Language-Based Approach to Protocol Implementation, PH.D Dissertation, University of Arizona, 1993.

[Al99] I. Ali, N. Al-Dhahir, and E.Hershey, J. *Predicting the Visibility of LEO Satellites*. IEEE Transactions on Aerospace and Electronic Systems (Oct. 1999) 1183-1190.

[AT87] American Telephone and Inc. Telegraph, *Unix System V Programmer's Guide,* Prentice Hall, Englewood Cliffs, NJ. 1987.

[AZ03] D.P. Agrawal, Q.Zeng, *Wireless and Mobile Systems*, Thomson Brooks/Cole Inc. 2003.

[BB02] M. Barbeau and F. Bordeleau *A Protocol Stack Development Tool Using Generative Programming,* The ACM/SIGSOFT Conference on Generative Programming and Component Engineering GPCE'02, Pittsburgh, 2002.

[BBD+98] M.Beck, H. Bohme, M. Dziadzka, U. Kunitz, R.Magmus and D. Verworner: *Linux Kernel Intervals,* Second Edition, Addison Welsley Longman, 1998.

[Bir00] K. Birman and et al, *The Horus and Ensemble Projects. Accomplishments and Limitations.* Proc. Of the DARPA Information Survivability Conference and Exposition (DISCEX'00). Hilton Head, South Carilina 2000.

[BNJ97] W. A. Beech, D. E. Nielsen and J. Taylor, *AX.25 Link Access Protocol for* Amateur Packet Radio, Version 2.2 November, 1997.

[CDP95] CDPD Forum, Inc. *CDPD System Specification: Release 1.1*. January 1995.

[CE99] K. Czarnecki and U. W. Eisenecker, *Generative Programming – Methods, Tools, and Applications*, Addison Wesley, 1999.

[CG97] J.Cai and D.J. Goodman, General Packet Radio Service in GSM, IEEE Communications, vol35, no, 10, pp.122-131, October 1997.

[CK87] M.Chepponis and P. Karn, *The KISS TNC: A Simple Host-to-TNC Communication Protocol*. ARRL 6th Computer Networking Conference, Redondo Beach, CA, 1987.

[Czar98] K. Czarnecki and et al, Generative Programming and Active Libraries Proceedings of the Dagstuhl-Seminar on Generic Programming, April 1998.

[DH95] S. Deering and R.Hinden, Internet Protocol, Version 6 (IPv6) Specification, RFC1883, December 1995.

[Dro99] A. Drozdek *Data Structures and Algorithms in Java*, Pacific Grove, CA, 1999.

[DWS02] S.Dawson, J.Wolff and J.Szielenski. *CHIPSat's TCP/IP Mission Operations Architecture - Elegantly Simple.* Proceedings of the 16th Annual AIAA/USU Conference on Small Satellites, Logan, Utah, 2002.

[Ela03] M.E. Elaasar, *XSTP: eXtended Satellite Transport Protocol,* Master Thesis, School of Computer Science, Carleton University, 2003.

[EZB+03] M.E. Elaasar, Z. Li, M. Barbeau and E. Kranakis, *The eXtended Satellite Transport Protocol: Its Design and Evaluation,* Proceedings of 17th Annual AIAA/USU Conference on Small Satellites, Logan, Utah, 2003.

[Hal96] F. Halsall Data Communications, Computer Networks and Open Systems, fourth edition, Addison-Wesley Publishing Company, 1996.

[HP91] N. C. Hutchinson and L. L. Peterson, *The x-kernel: An architecture for Implementing Network Protocols*, IEEE Transaction on Software Engineering, 17(1): 64-76, January 1991.

[HPM+00] H. Heidt, J. Puig-Suari, A. Moore, S. Nakasuka and R. Twiggs, CubeSat: A New Generation of Picosatellite for Education and Industry Low-Cost Space Experimentation, Proceedings of the Thirteenth Annual AIAA/USU Small Satellite Conference, Logan, UT, 2000.

[Igl] IglooFTP Pro, available online [accessed on April 23, 2003] http://www.linux.org/apps/AppId_3685.html .

[Jam96] Jamalipour, A. *Low Earth Orbital Satellites For Personal Communication Networks.* Artech House Publishers, 1996.

[JMH+01] D.B. Johnson, D.A. Maltz, Yih-Chun Hu and Jorjeta G. Jetcheva, The Dynamic Source Routing Protocolfor Mobile Ad Hoc Networks (DSR), IETF MANET Working Group, Internet Draft, November 2001.

[KK95] M.Khan and J. Kilpatrich, *MOBITEX and Mobile Data Standard, IEEE* Commun. Mag., vol.33, no.3, pp.96-101, March 1995.

[KT75] L.Kleinrock and F.A.Tobagi, *Packet Switching in Radio Channles: Part I – Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics.*IEEE Transactions on Communications, Vol. COM-23, No. 12, Dece,ber 1975, pp. 1400-1416.

[MA01] J.O.Mitchell, A. Samuel, *Advanced Linux Programming*, New Riders, 2001.

[Mal00] J. Malinen, *Using Private Addresses with Hierarchical Mobile IPv4*, Master Thesis, Department of Computer and Engineering, Helsinki University, 2000.

[MB92] C. Maeda and B.N. Bershad, *Networking Performance for Microkernels*, In Preceedings of Third Work Shop on Workstation Operating Systems, WWOS-3, April, 1992.

[MBK+96] K. McKusick, K. Bostic, M. J. Karels and J.S. Quarterman, *The Design and Implementation of 4.4BSD Operating System.* Available online [Accessed on April 23, 2003] Addison-Wesley Longman, Inc, 1996. http://www.freebsd.org/doc/en_US.ISO8859-1/books/design-44bsd/ .

[MHS+03] W. Marchant, M. Hurwitz, M.Sholl and E.R.Taylor, *Status of CHIPS: A NASA Univeristy Explorer Astronomy Mission*, 2003.

[Mos96¹] D. Mosberger, *Message Library Design Notes*, Network System Research Group, Department of Computer Science, University of Arizona, January 1996.

[Mos96²] D. Mosberger, *Map Library Design Notes*, Network System Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

[MP96] Mars Pathfinder web page available online [accessed on April 24, 2003] at http://mars.jpl.nasa.gov/MPF/, 1996.

[NcF] NcFTP software, available online [accessed on April 24, 2003] http://www.ncftp.com/ncftp/

[NSR96] Network System Research Group, Department of Computer Science, University of Arizona, *x-kernel Programmer's Manual (Version 3.3),* January 1996.

[NUS] National University of Singapore, Department of Electrical and Computer Engineeing, available online [accessed on April 24, 2003] http://www.ee.nus.edu.sg/

[PD96] L.L. Peterson and B.S. Davie, *Computer Networks: A System Approach*, Morgan Kaufmann Publishers, San Fransico, CA, First Edition, 1996.

[PR85] J. Pastol, J.Reynolds, *File Transfer Protocol*, RFC959, October 1985.

[Pro] ProFTPD, available online [accessed on April 23, 2003] http://www.proftpd.org/, updated on Dec 28,2002.

[PW01] D. Petriu and M. Woodside, *Incorporating Performance Analysis in the Early Stage of Software Development Using Generative Programming Principles* in ECOOP 2001.

[PW90[1]] H.E Price, and J.Ward, *PACSAT Protocol Suite – An Overview*, ARRL 9th Computer Networking Conference, pp. 203-206, August 1990.

[PW90[2]] H.E.Price, and J.Ward, *PACSAT Broadcast Protocol*, ARRL 9th Computer Networking Conference, pp. 232-238, August 1990.

[RM95] J. Radzik, and G. Maral, *A methodology for rapidly evaluating the performance of some low Earth orbit satellite systems*. IEEE Journal on Selected Areas in Communication. (Feb. 1995), 301-309,1995

[SG98] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, Fifth edition, AddisonWesley Inc., 1998.

[SCP98] Space communication protocol specification (SCPS): Rationale, requirements and application notes. Technical report, Consultative Committee for Space Data Systems (CCSDS), August 1998. Green Book, Daft 0.4.

[Ste98] W. R. Stevens, *UNIX Network Programming*. Prentice Hall PTR, Upper Saddle River, NJ, 1998

[TNM+93] C.A. Thekkath , T.D Nguyen, E. Moy and E.D. Lazowska, *Implementing Network Protocols at User Level*, in ACM SIGCOMM'93 1993.

[Tro] TrollFTPD1.27, available online [accessed on April 24, 2003] http://linux.bankhacker.com/en/software/Troll-FTPd/ .

[Vel99] T. Veldhuizen, *C++ Templates as Partial Evaluation*, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99).

[Win71] J.M. Winett, *Definition of a socket*, RFC0147, May-07-1971.

[WP90] J.Ward, and H.E. Price, *PACSAT Protocol: File Transfer Level 0*, ARRL 9th Computer Networking Conference, pp. 209-231, August 1990.

[WSJ02] G. J. Wells, L. Stras, and T. Jeans, *Canada's Smallest Satellite: The Canadian Advanced Nanospace eXperiment (CanX-1)*, Small Satellite Conference, Logan, Utah, 2002.

[WuF] WuFTP home page available online [accessed on April 24, 2003] at http://www.academ.com/academ/wu-ftpd/, updated on Jul 6, 1998.

[Xu01] Y. Xu, *Implementation of Location Detection, Home Agent Discovery and Registration of Mobile IPV6,* Master Thesis, School of Computer Science, Carleton University, 2001.

[Zha02] S. Zhang, *Channel Management, Message Representation and Event Handling of a Protocol Implementation Framework for Linux Using Generative Programming*, Master Thesis, School of Computer Science, Carleton University, 2002.