2nd Annual Conference on
Communication Networks and Services Research (CNSR 2004)
Fredericton, N.B., Canada, May 19-21, 2004

Performance of Generative Programming Based Protocol Implementation

Zheyin Li and Michel Barbeau
School of Computer Science, Carleton University,
1125 Colonel By Drive
Ottawa (Ontario), Canada K1S 5B6
{zheyin99@yahoo.com, barbeau@scs.carleton.ca}

Abstract

Protocol Implementation Framework for Linux (PIX) is a protocol development tool using generative programming. It aims at capturing the similarities in behaviors among different layers of protocols and grouping solutions to cross-cutting concerns of communication systems. It achieves a high degree of configurability by providing several combinations which could be chosen to generate desired protocols. This paper addresses the following open question. How does the performance of generative programming based protocol implementation compare with traditional protocol implementation techniques? This paper 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 PIX and generative programming with NcFTP, which uses structured programming, and x-Kernel, which uses structured and object-based programming.

1. Introduction

The strategies for network protocol implementation could be categorized into three approaches: kernel level, user level or mixed. This classification is according to the placement of the code of the implementation 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 shows superiority in code development and maintenance, which are relatively easier. The performance could be, however, the biggest concern. A mixed approach is a compromise of these two strategies.

Implementing a network protocol 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 [3], BSD Unix [8] and System V Streams [1] are some well-known protocol implementation frameworks. Protocol Implementation Framework for Linux (PIX) [2] is a framework based on generative programming (GP) [5]. GP is as a technique to build models for families of systems and then generate concrete 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 cross cutting concerns of different protocols so that developers can assemble various protocols to satisfy different needs. Commonalities and variations of 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 [6]. PIX reuses some of the key concepts of x-Kernel, e.g. Protocol and Session. PIX and x-Kernel share similar concepts, while the design goal 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 - currently it could provide 48 combinations of ways to assemble a protocol. Providing configurability is not only the utmost difference between PIX and x-Kernel, but it is also a difference between PIX and the other aforementioned protocol development frameworks. The Horus and Ensemble [4] 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 on protocol structure and optimization instead of functionality. The protocols generated from PIX are of a well-defined prototype and specific functionalities need to be added to this prototype in order to make it become a complete network protocol.

To the best of our knowledge, PIX is the first attempt to use GP in network protocol development. PIX was designed to make the protocol development work easier, but not at the expense of correctness or performance. Zhang [9] provides partial performance evaluation of PIX. It compares 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.

This paper 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. Latency and throughput are the metrics used to compare the performance of file transfer. Latency is defined as the time required to transfer a file from one end of the network to the other, e.g. the download of a file from a server to a client. The latency is comprised of several parts: the time to open and close a TCP connection for data transfer; the time to send a file transfer request, replies and the 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. From a protocol point of view, the transmission time, which depends on network bandwidth, is beyond control, so the part that varies is the processing delay, which is what we are investigating. Throughput is a measure of how much data can pass through a channel per unit of time. It depends fundamentally on available 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. They are key indicators of the performance of a network.

The contribution of this paper is a performance comparison of a PIX based FTP with an x-Kernel based FTP and Linux NcFTP over Internet sockets.

This paper is structured as follows: Section 2 gives an overview of protocol implementation in PIX. Section 3 gives a quantitative performance evaluation of the FTP-PIX, namely, in latency, throughput and resource usage. Section 4 concludes the paper.

2. Protocol implementation in PIX

This section gives an overview of protocol implementation in PIX, contrasts it with x-Kernel and discusses the design of an implementation of FTP in PIX and x-Kernel.

2.1. Overview of PIX

PIX is designed to capture the 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 family of systems, and belongs to domain engineering [5]. Domain engineering considers development for reuse and consists of three basic steps. Firstly, domain analysis involves domain scooping and feature modeling. Domain scooping 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 1, according to [2]. The *Protocol* concept, the root in the diagram, captures two functions. Firstly, it implements the algorithms that make up a unit of communication functionality. For example, an IP protocol object, which adds, strips and processes IP headers and routes messages to their right destination. It is also responsible for managing *Sessions*, which represent end points of network communication channels. A Protocol 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 1, 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).

A detailed description of features in the diagram follows. *Session* is an instance of a network connection in Protocol. It holds the information for the connection. It is created at runtime either actively, when the application initiates the communication, or passively, when higher level protocol informs its lower level protocol of its readiness to accept connections. *Participant* identifies the parties that communicate with each other in a network. *Message* is a two-part structure that separates the header from the user data. There are two basic operations for message processing, prepending headers to outgoing messages and stripping headers from incoming messages. There are two alternatives: *SingleBufferMessage* and *BufferTreeMessage*. *Event-*

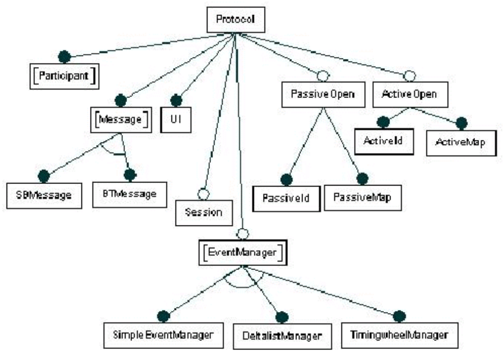

Figure 1. Feature diagram of PIX.

```
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. GenVoca representation of the PIX architecture.

Manager 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 set timeouts and act on messages that have not been acknowledged or perform periodic maintenance functions. There are three variations: *SimpleEventManager*, *DeltaListManager* and *TimingWheelManager*. *Uniform Interface (UI)* defines the common operations to all protocols and sessions. Each protocol or session 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, it 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 (like 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 accept 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, also features are mapped to components. GenVoca grammar as a widely used methodology to capture the system architecture in a hierarchical layered fashion is utilized here. Figure 2 gives the GenVoca representation of the PIX architecture in seven layers. The Message and EventManager features 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 (L1) 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 protocols with sessions that only act as client. And the third option ProtocolUI is used by protocols that do not embed sessions.

The last step - domain implementation turns components into C++ classes. A component takes a component from the layer below it as its parameter, which means components are implemented as parameterized classes, or be more specific 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 a wanted protocol. To assemble a protocol, there are two ways: a manual assembly and an automatic assembly. Manual assembly means developer writes his/her own configuration repository for each protocol in this family, which is tedious and restrictive. A more efficient and practical way is the auto generation of system by providing its specification to a generator. A generator is a class tem-

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

Figure 3. A specification example provided to the protocol generator.

Framework	x-Kernel	PIX
Approach	Object-based	Object-oriented and generative Programming
Language	C	C++

Table 1. Comparison between x-Kernel and PIX.

plate 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 components into a system [5]. Detailed explanation of how protocol generator is implemented could be found in [9]. Figure 3 shows an example of a specification to generate a protocol from the protocol generator. It generates an FTPBase base class 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 described in the next section.

2.2. Comparison of PIX and x-Kernel

PIX reuses the key concepts defined in x-Kernel, while these two protocol development frameworks do differ one from the other. The key distinction of PIX is a high degree of configurability by combining well-established techniques of protocol implementations. Table 1 summarizes and compares the characteristics of these two frameworks. Because PIX utilizes the GP model it provides choices for features such as message representation.

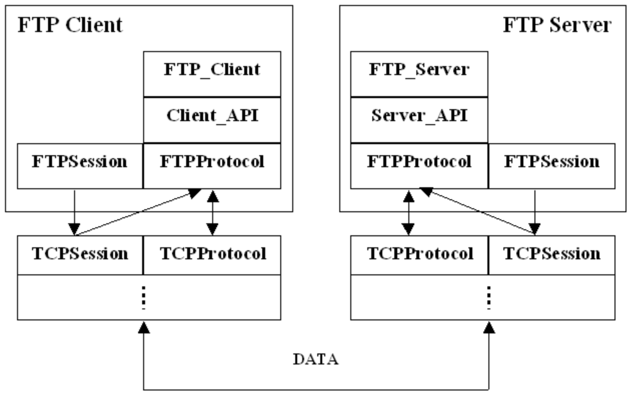

Figure 4. The FTP architecture in PIX.

The protocols generated from PIX are already able to perform the basic behavior, like sending/receiving/demultiplexing messages. What developers need to do is to add protocol specific algorithms or rules to get a fully functional protocol. In contrast, in x-Kernel developers need to develop each component of a protocol, in conformance to a uniform protocol interface (UPI) to ensure the interoperability with other layers. To be more specific, each protocol from x-Kernel needs to implement its own active open, passive open, close and control connections, demultiplex, send and receive message functions. However, a protocol generated from PIX does not need to implement any of those operations unless it requires special processing.

2.3. Design of an implementation of FTP in PIX and x-Kernel

Figure 4 illustrates the architecture of the FTP developed in PIX. There are a client protocol and a server protocol. Each part contains three sub layers. The top layer *FTP_Client* or *FTP_Server* contains the 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, *Client_API* handles FTP client commands and *Server_API* consists of sever handlers for these commands. The bottom layer is composed of *FTPProtocol* and *FTPSession* that takes care of real data transmission and interaction with the transport layer, e.g. *TCPProtocol* and *TCPSession*.

The class diagram of the PIX implementation of FTP is depicted in Figure 5. Each class is represented as rectangle frame with the class name on the top, followed by the

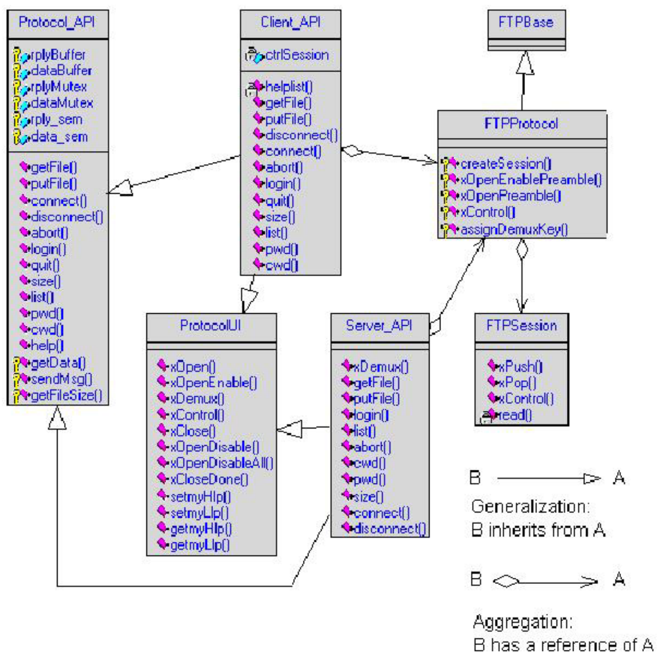

Figure 5. Class diagram.

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.

ProtocolUI is a class defined in 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. The *xDemux()* method is used by a low-level session to pass messages to its 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()* is used to cancel all *xOpenEnable()* operations. The *xCloseDone()* is used by a low-level protocol to inform the high-level protocol that a peer participant has closed the session that was originally opened by the high-level protocol. The *getmyHlp()* and *getmyLlp()* are used to get a protocol's high-level and low-level protocols. The *setmyHlp()* and *setmyLlp()* are used to set a protocol's closely related high-level and low-level protocols, which is mainly used to build a protocol stack graph.

FTPBase is produced by PIX's protocol generator, as shown in Figure 3 and it conforms to the protocol interface defined in ProtocolUI.

FTPProtocol is inherited from FTPBase and 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 a TCP or an IP protocol whose key is embedded 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 xOpenEnable() operation of FTP, the reference count of the enable object is incremented. The *createSession()* method creates an FTP session. The *xOpenPreamble()* calls its low-level protocol's *xOpen()* to get the low-level session, save this session as the active key. The *xOpenEnablePreamble()* calls its low-level protocol's *xOpenEnable()*. The *assignDemuxKey()* method saves the session in the parameter list as the active key and it 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 represents an FTP channel between the client and server. When a message is to be sent out, the *xPush()* method of FTPSession is called, while a message is received, the *xPop()* method is called. The receiving FTP session has the responsibility to read 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 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 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. They are pure virtual functions. Its 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 implements two interfaces of different nature, one for the protocol stack, the other is for FTP server functionality. Therefore it inherits from both ProtocolUI and Protocol_API to provide service to server application. It decouples server's services from its concrete implementation. It overrides the public methods in *Protocol_API* to define server's response to user commands.

Client_API also provides two interfaces, one for the protocol stack, the other for FTP client functionality. 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.

As discussed earlier, PIX borrows the fundamental ideas of 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 is, however, written in a structured programming language, i.e. C, it is object-based and it provides its own object infrastructure, which becomes the glue to make it possible for 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.

3. Performance evaluation

This chapter reports on an experiment conducted to evaluate the performance of PIX. The objective is to quantify the impact that PIX and generative programming have on protocol performance. The experiment aims to compare the performance of a file transfer protocol implemented in PIX with an x-Kernel implementation and a built-in Linux implementation. Firstly, we measure the latency and throughput of the file transfer protocol implementations. Secondly, the CPU and memory usage are measured.

3.1. Experimental environment

The equipment used in the experiment is a pair of Intel Pentium III, 547 MHz and 192 MB PCs. 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 6. 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 built-in Linux experiment, the NcFTP implementation is used. The x-Kernel implementation is interfaced to the Socket API, also using a similar Adapter protocol.

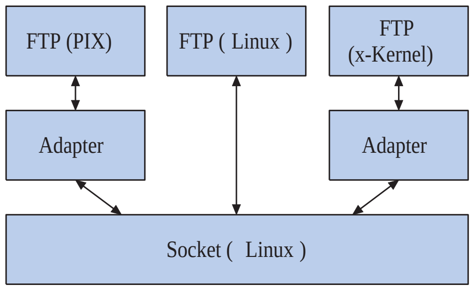

Figure 6. Protocol graph of the PIX implementation, built-in Linux implementation and x-Kernel implementation.

The experiment consists of running the same test for 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 to be noticed 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 connection establishment and control information exchange. Taking all the overhead into account is important in our experiment because we are concerned about the impact of the whole protocol stack on performance. Therefore, we consistently define in PIX, NcFTP and x-Kernel the time it takes to retrieve a file 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.

3.2. Latency

Figure 7 gives the results for transferring files of size 1 B to 100 MB. Horizontal axis is log_{10} of file size in bytes while vertical axis is log_{10} of time in milliseconds. Figure 8 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 difference tends, however, to be not noticeable as the size of files grows.

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

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

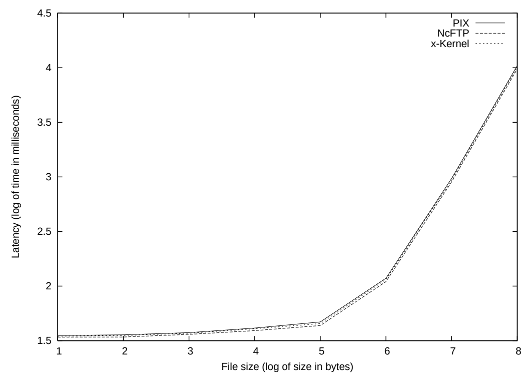

Figure 7. Latency of file transfer in PIX, NcFTP and x-Kernel (log_{10}-log_{10} scale).

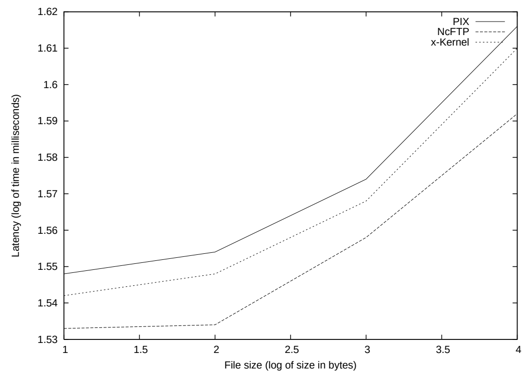

Figure 8. Latency of transfer of files under 100 KB (log_{10}-log_{10} scale).

where T_{NcFTP} is the latency of NcFTP and T_{PIX} is the latency of PIX. The value of p oscillates between 3.5%, smallest file, to 7.2%, largest file. In the equation, if we substitute the latency of NcFTP T_{NcFTP} by the latency of x-Kernel $T_{xKernel}$, then the value of p oscillates between 1.5%, smallest file, to 2.8%, largest file. It is a measure of how much PIX is slower compared to NcFTP and x-Kernel.

3.3. Throughput

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 2. All three implementations are capable of saturating the 100 Mbps net-

S	PIX	NcFTP	x-Kernel
1 KB	0.208	0.216	0.211
10 KB	1.89	2	1.92
100 KB	16.62	17.92	17.21
1 MB	67.8	72.73	69.57
10 MB	82.9	88.89	85.38
100 MB	76.55	82.05	78.66

Table 2. Throughput, in M bps, of file transfer in PIX, NcFTP and x-Kernel.

work. 100% efficiency is not possible because of the link layer overhead.

The throughput of transferring files of size 100 KB and under is low because the network pipe cannot be fully utilized. It is also shown in the table 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 access, which reduces the system throughput.

3.4. Resource usage

Resource consumption is an important performance criterion. it refers to the CPU and memory usage of a program execution.

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 retrieve procedure. This experiment is repeated 50 times and the CPU time usage is averaged. Figure 9 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 realtime, we use the *top* command of Linux which returns the dynamic information regarding the memory usage of a specific process. Table 3 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 ready for processing a request. During the procedure of transferring a file, the memory usage increases and 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,

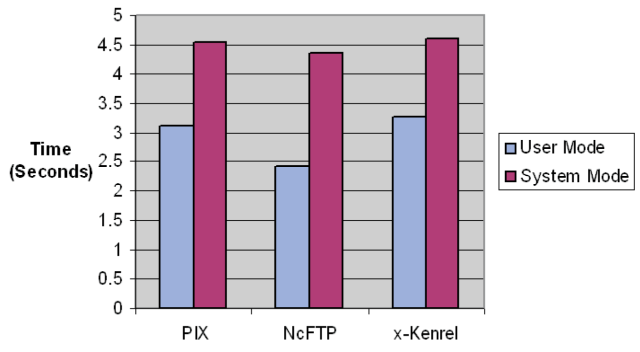

Figure 9. CPU time usage.

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

Table 3. Footprint, in KB, of file transfer in PIX, NcFTP and x-Kernel.

the bigger the increase. The column under the field of *100 MB File Transfer* shows the peak value of memory usage, which is during the transfer a 100 MB file. The NcFTP software occupies more memory because it supports complete FTP functionalities, while FTP of PIX implements part of them and FTP of x-Kernel only supports the file transfer capability.

4. Conclusion

In a companion paper [2], it is argued that, with respect to more traditional protocol development tools, PIX offers configurability superiority because of the use generative programming. An important issues 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 paper has analyzed the performance of PIX and GP based protocol implementation with respect to traditional approaches.

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 2). Relating to resource usage, CPU usage time is of the same order with all three implementations. Memory usage by NcFTP is higher, but is currently support more functionality.

From the statistics of our experiments, we could state that using generative programming adds very light additional measurable cost with respect to object-based or structured programing when used for protocol implementation. Take into consideration of the benefit of a high degree of configurability, generative programming is a very promising approach of implementing protocols.

Acknowledgment

The authors graciously acknowledge the financial support received from the following organizations: Natural Sciences and Engineering Research Council of Canada (NSERC) and Mathematics of Information Technology and Complex Systems (MITACS).

References

[1] American Telephone and Inc. Telegraph: Unix System V Programmer's Guide. Prentice Hall, Englewood Cliffs, NJ. (1987)

[2] Barbeau and F. Bordeleau: A Protocol Stack Development Tool Using Generative Programming. in: D. Batory and C. Consel (Eds.): Proceedings of Generative Programming and Component Engineering (GPCE). Lecture Notes in Computer Science 2487. (2002)

[3] Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D.: Linux Kernel Internals - Second Edition. Addison Wesley Longman. (1998)

[4] Birman, K., Constable, R., Hayden, M., Kreitz, C., Rodeh, O., van Renesse, R., Vogels, W.: The Horus and Ensemble Projects: Accomplishments and Limitations. Proc. of the DARPA Information Survivability Conference and Exposition (DISCEX '00). Hilton Head, South Carolina. (2000)

[5] Czarnecki, K. Eisenecker, U.W.: Components and Generative Programming. Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on Foundations of Software Engineering. (1999) 2–19

[6] Hutchinson, N.C., Peterson, L.L.: The x-Kernel: An Architecture for Implementing Network Protocols. IEEE Transactions on Software Engineering. **17** (1) (1991) 64–76

[7] Li, Z.: Performance of Generative Programming Based Protocol Implementation. Master Thesis, School of Computer Science, Carleton University. (2003) (Available at: www.scs.carleton.ca/∼barbeau/Thesis/li.pdf)

[8] Wright, G.R., Stevens, W.R.: TCP/IP Illustrated - Volume 2. Addison Wesley. (1995)

[9] Zhang, S.: 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) (Available at: www.scs.carleton.ca/∼barbeau/Thesis/zhang.pdf)