Channel Management, Message Representation and Event Handling of a Protocol Implementation Framework for Linux Using Generative Programming

By
Song Zhang

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Science
in Information and Systems Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

June 3, 2002

© Copyright
2002, Song Zhang

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79805-4

Canada

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

Channel Management, Message representation and Event
Handling of a Protocol Implementation Framework for
Linux Using Generative Programming

Submitted by
Song Zhang

in partial fulfillment of
the requirements for the degree of
Master of Science
Information System Science

Dr. Frank Dehne
(Director, School of Computer Science)

Dr. Michel Barbeau
(Thesis Supervisor)

Carleton University
June 3, 2002

Abstract

There exist two main approaches for the implementation of protocols: the user-level approach and the kernel-level approach. An implementation of protocols at the user level runs as a user process while a kernel-level implementation runs in the operating system. The Protocol Implementation Framework for Linux (PIX) is a tool for developing a user level implementation of protocols by means of Generative Programming (GP). It provides a set of libraries and uniform interfaces for a client-configurable protocol stack implementation.

GP focuses on software system families. It uses concepts to represent the elements with variation points, and features to model the configurable aspects of a concept. Generators in GP take the specification of a system and automatically manufacture the one required. In this thesis, session management, message representation and event handling in PIX are developed to enhance its functionality. A passive open feature for session management provides a local node with the ability to accept connections. The creation of a channel is completed as soon as a remote node requests a connection. The protocol generator and session generator are designed to assemble implementation components automatically with previously-defined features, as well as new features, which are defined in this thesis. Three event-handling features have been developed for use in different situations. SimpleEventManager handles a few events. DeltalistManager can schedule more events efficiently. TimingwheelManager retains the efficiency of DeltalistManager while optimizing the event storage process. The BufferTreeMessage feature has been developed to allocate messages dynamically and avoid time-consuming data copying.

Acknowledgments

First of all, I would like to express my sincere thanks to Professor Michel Barbeau, my supervisor, for his academic and technical guidance and the valuable and profound discussions. His concern and assistance are greatly appreciated.

I want to thank those people on the team, Mr. Amir Ghavam, Mr. Greg Campbell, Ms. Hadeel Kassabi, Ms. Honghui Luo, Ms. Huan Qi, Ms. Lan Lu, Ms. Maoning Wang, Mr. Maged Ellasssar and Ms. Zheyang Li for their beneficial presentations and discussions.

This thesis is dedicated to my wife, Tianyu Yang, for her understanding and support during the whole period of this study, and to my lovely daughters, Yue and Lin, for the happiness and relaxation they have brought to my life.

Table of Contents

Abstract	iii
Acknowledgments	iv
Table of Contents	v
List of Tables	vii
List of Figures	viii
1. Introduction	1
1.1 Motivation	1
1.2 Thesis Contributions	4
1.3 Thesis Outline	6
2. Review of Literature	7
2.1 Network Support in Current Operating Systems	7
2.2 The X-Kernel	11
2.3 The P-Kernel	13
3. Generative Programming for Telecommunications Protocols	14
3.1 Generative Programming	14
3.2 Protocol Implementation Framework for Linux (PIX)	15
3.2.1 Domain Analysis	15
3.2.2 Domain Design	21
3.2.3 Domain implementation	25
4. New Features of PIX.....	29
4.1 Overview of New Features	29
4.2 The PassiveOpen Feature	32
4.3 The Generator	38

5. Event Feature	46
5.1 Introduction	46
5.2 The SimpleEventManager Feature	48
5.3 The DeltalistManager Feature	55
5.4 The TimingwheelManager Feature	65
5.5 Performance Evaluation	73
 6. Message Feature	 76
6.1 Introduction	76
6.2 The SingleBuffMessage Feature	77
6.3 The BufferTreeMessage Feature	77
6.3.1 Concepts	77
6.3.2 Data Structure	78
6.4 Performance Evaluation	87
 7. Conclusions and Future Work	 89
7.1 Conclusions	89
7.2 Future Work	90
 Bibliography	 92

List of Tables

Table 5.1 Operations for managing events	48
Table 6.1 Performance evaluations	88

List of Figures

Figure 2.1	Internet architecture	8
Figure 2.2	Implementation of socket interface	9
Figure 3.1	Feature diagram of the PIX.....	18
Figure 3.2	Process of managing a connection between a client and a server....	19
Figure 3.3	Sequence diagram in an active open	20
Figure 3.4	Sequence diagram in an active close	21
Figure 3.5	Component categories for the protocol	22
Figure 3.6	Derivation of a layered architecture (a), (b), (c)	23
Figure 3.7	GenVoca grammar for the protocol family	24
Figure 4.1	Extended feature diagram of the PIX	31
Figure 4.2	Sequence diagram in a passive open	33
Figure 4.3	Session creating process in a passive open	34
Figure 4.4	Sequence diagram in a passive close	35
Figure 4.5	Extended GenVoca grammar for the protocol family	36
Figure 4.6	Stages of a configuration generator	39
Figure 5.1	Event flow in the SimpleEventManager	48
Figure 5.2	Delta list implementation of an event manager	55
Figure 5.3	Event flow in the DeltalistManager	56
Figure 5.4	Inserting a new event into a delta list	61

Figure 5.5	Timing wheel implementation of an event manager	66
Figure 5.6	Event Flow in the TimingwheelManager	66
Figure 5.7	Inserting a new event into an array of lists.....	73
Figure 5.8	Performance comparison of the three event managers	74
Figure 6.1	Message extension process of an overflow	78
Figure 6.2	Class diagram of Node, PairNode and LeafNode	79
Figure 6.3	Class diagram of the BufferTreeMessage feature	82

Chapter 1

Introduction

1.1 Motivation

Network software makes the computer's resources available to other network nodes and provides the data and control abstractions for passing information from process to process over the network. Some of the current operating systems (OSs) [17,18] have their network software built in as subsystems, while others have their network software built in as a user process. Some of them are open source, which means the network software can be extended to allow for the integration of new protocols. They are useful in developing a new set of protocols for new types of communications, such as mobile and wireless communications. They provide an environment for research and allow for the testing of new protocols at the physical, link, or network layers, or even higher.

The X-Kernel

The x-kernel [1-6], developed at University of Arizona, is an example of this type of experimental environment for the implementation of protocols. It is an application framework that is composed of a set of data structures, a library of common operations, and a model of execution. The client software just needs to hook its special protocols

onto the framework and actualize the framework to obtain a customized application. Use of a framework accelerates the development of protocol architectures.

There are three main types of abstract objects in the x-kernel: protocol, session, and message. A protocol object is a static entity, created in the application program, modeling a given protocol. Protocols are composed of sessions that are created dynamically and model end-points of communication channels. Messages are created dynamically and flow from one protocol or session to another. A message is usually composed of headers and data.

An important concept within the model of the x-kernel is the Uniform Protocol Interface (UPI). The communication primitives supported by protocols and sessions are uniform and generic across all the layers. They are actualized using a function pointer mechanism. The UPI achieves a uniform structure from one layer to another that facilitates the readability of a protocol stack to a considerable extent.

There are two characteristics to the x-kernel. First, it is implemented in the C language. Second, it is aimed at supporting most of the protocols and operating systems, thus leading to a huge collection of operations, data structures and status parameters. It might be hard to use the x-kernel in conditions where resources are limited.

Protocol Implementation Framework for Linux (PIX)

Small and efficient network software is required within the special environment of mobile and wireless communications, which usually has low and limited power, few resources and low memory. The PIX in C++ is an example of this user space software

that is being developed at Carleton University specifically to provide the experimental environment for protocol implementation in mobile and wireless communications.

Like the x-kernel, the PIX provides the thread-per-message model and several supporting libraries, such as the map library, message library, and event library. But more importantly, the PIX is developed by means of the Generative Programming (GP) approach and has better configurability. GP focuses on a family of software systems. It defines concepts for any elements and structures in the domain of interest, as well as features to represent the distinguishable characteristics of concepts. In PIX, the Protocol concept is defined to represent the specific communication rules. The various properties of the Protocol concept are represented as its features. The ActiveOpen feature models the session management of the protocol and provides the PIX with the functionality of a client. A Uniform Interface (UI) feature defines the common operations among the protocols and sessions for the clarity of the implementation of the protocol stack. A SingleBuffMessage feature is used when the whole memory of a message can be pre-allocated.

Compared with the x-kernel and the p-kernel, the PIX has better configurability. It provides several possible configuration repositories. Clients can select various feature combinations in order to construct specifically customized and optimized programs.

1.2 Thesis Contributions

The development of the PIX is currently carried on by a research group, aiming to realize the communications among LEO satellites and ground stations. My responsibility in this group is to implement several features for PIX to enhance its functionality, while the other members are working on the implementation of protocols, such as Simple Network Management Protocol (SNMP) [26], File Transfer Protocol (FTP) [27], Satellite Transport Protocol (STP) [28], Dynamic Source Routing (DSR) [29], IPv6 [30] and a wireless LAN driver.

The previous version of PIX does not have the support for server functionality. The implemented components can only be manually assembled. There is only a simple event manager for event handling, which might be not appropriate when the time is critical. The single buffer message model for message representation in the PIX might not be efficient when the size of messages can not be predetermined.

Considering the problems above, this thesis provides relevant solutions. As for the lack of support of server functionality, the PassiveOpen feature of the Protocol concept is created. Three generators, a protocol generator, a Uniform Interface (UI) generator and a session generator are designed and implemented to assemble the implementation components automatically. Three types of event managers, SimpleEventManager, DeltalistManager and TimingwheelManager are implemented as features of the Protocol concept for different requirements. A BufferTreeMessage feature of the Protocol concept is implemented for the situation when the size of messages can not be predetermined.

The PassiveOpen feature provides the PIX with the complete procedures for the creation and management of channels. This is necessary when a local node (such as a server) is willing to accept connections from remote nodes. When a remote node has finished the transmission of data and releases the connection, the local node issues a passive close of the related channels from the bottom of the protocol stack up to the top.

The protocol generator, UI generator, and session generator are very important in developing the GP model for the PIX. They allow a PIX user to assemble the implementation components automatically without having to worry about writing each configuration repository manually.

Three event-handling features are implemented in C++ for the PIX. The SimpleEventManager is appropriate if there are only a few outstanding events to be handled and the time is not critical. The DeltalistManager and the TimingwheelManager behave better when there is a high number of outstanding events to be handled. We show that the TimingwheelManager is the most efficient in scheduling new events.

The SingleBuffMessage feature is used when the size of messages can be predetermined, since it performs better because all the memory is pre-allocated. However, if the size of messages can not be predetermined (such as, the length of the source routing extension header in IPv6, or the lengths of some header options in IPv4), the BufferTreeMessage feature should be used. It has the ability to allocate memory for messages dynamically, while time-consuming data copying is avoided.

1.3 Thesis Outline

The thesis has the following structure and aims to construct some fundamental components of the PIX.

Chapter 2 gives an overview of the network support in current OSs, the x-kernel framework and the p-kernel framework.

Chapter 3 examines the application of generative programming in telecommunications protocols. Generative programming and a protocol implementation framework for Linux using generative programming are introduced.

Chapter 4 presents the new features added to the PIX. The passive open feature is discussed in detail. In addition the generator, which is an important methodology for manufacturing the implementation components automatically, is introduced. The protocol generator, UI generator, and session generator are explained.

Chapter 5 considers event handling, which is one of the most important issues in the implementation of protocols. Three features of event managers - SimpleEventManager, DeltalistManager, and TimingwheelManager - are discussed.

Chapter 6 explains the data structures, classes and functionality of the BufferTreeMessage feature in detail. The SingleBuffMessage feature is discussed for purposes of comparison.

Chapter 7 concludes the thesis and discusses the future work.

Chapter 2

Review of Literature

2.1 Network Support in Current Operating Systems

Almost all of the current operating systems (OSs) support computer network communications by implementing the Internet architecture (also called the TCP/IP suite) and/or other protocol suites as part of the kernel [8-10,17,18]. Here we will focus only on the TCP/IP suite, as it is the dominant one at present. The TCP/IP suite [8-10] defines four layers of network protocols, where one or more protocols implement the functionality assigned to a given layer and the strict methods of communication between them. Figure 2.1 depicts the Internet architecture. An implementation of a protocol may communicate only with a peer entity speaking the same protocol at the same layer, or with the service interface of a protocol in the layer above or below in the same system. There is no strict layering, so programmers are free to define new channels or applications to run on top of any of the existing protocols.

Figure 2.1 Internet architecture

User processes communicate with network protocols via application programming interfaces (APIs). Applications can easily be ported from one OS to another through the use of APIs. The socket interface is one example of a widely supported API.

The socket interface [8,9,18] can be thought of as the point where a local application process makes contact with the network. The socket interface hides three layers: the socket layer, the protocol layer and the driver layer. All these layers are inside the kernel. Figure 2.2 shows the implementation of the socket interface. The socket layer consists of a number of system calls needed to create sockets, attach the sockets to the network, send and receive messages through the sockets, and close the sockets. The driver layer deals with transmitting the data. The protocol layer handles the detailed processing of system calls. Some approaches to the implementation of the protocol layer are discussed below.

Figure 2.2 Implementation of the socket interface

D. E. Comer [11] takes the process-per-protocol model for the implementation of protocols embedded inside the OS kernel. The processes for IP, TCP input, TCP output and TCP timer management execute independently. Queues and ports are used to pass messages among the protocols. A semaphore mechanism helps to synchronize the execution of processes. However, a problem with this model is that a context switch is required at each level of the protocol graph to pass a message from one layer to another, which is time consuming and inefficient.

W. R. Stevens [12] implemented the TCP/IP protocol layer in the BSD 4.4 OS kernel. In the implementation, each protocol defines its own routines. A fundamental memory buffer, MBUF, is used throughout the networking code to hold and transfer data, headers and other information among the protocols. There are four different types of MBUF: two are small – 128 bytes – and are used for dealing with headers and a few bytes of

information; two others (called CLUSTERS) are large – 2048 bytes – and are used for dealing with large amounts of data. Several MBUFs are chained together as a list if needed, for the storage and transfer of messages. MBUFs were designed at a time when memory was an expensive resource that needed to be allocated carefully. They still form the basis for all the Berkeley-derived networking code in use today.

The Linux kernel [17,19] supports the BSD socket interface for application programs, the TCP/IP protocol suite for Internet, and many low-layer protocols like Ethernet, FDDI and ATM. The Linux protocol stack is a part of the kernel. The most important aspect of the Linux protocol implementation is the use of SKBUF buffer management for data passing among protocols. SKBUF is different from the MBUF originated from BSD. Unlike MBUF where a small memory block is used, SKBUF uses continuous memory which is big enough to store packets as a whole, and which is allocated by calculating the maximum amount of space required by a message. The space includes the data part and the length of all the headers of the various layers across the protocol stack. SKBUF is more efficient than MBUF since it does not need to manage the links among the small buffers in MBUF.

The difficulty of implementing protocols in the kernel of an OS is that whenever a new protocol is designed and implemented, the kernel has to be re-linked to support it. It is expensive to test new network protocols in this way. Even worse, some current OSs, like QNX and Microsoft Windows, are not open source, thus making the testing and implementation of new protocols difficult in an academic context.

2.2 The X-Kernel

The x-kernel [1-6], a user-level implementation of protocols in C as libraries embedded in the application programs, was developed at University of Arizona in 1989. It is an experimental software framework designed specifically for the implementation and testing of network protocols. It provides an infrastructure that makes it easy to implement network protocols. This infrastructure is general enough to support a variety of protocols. In contrast, many existing OSs are either built around a fixed set of protocols or do not provide enough explicit infrastructure to perform protocol implementation.

One of the key notions in the x-kernel is Uniform Protocol Interface (UPI), which is a common interface between adjacent protocol layers. Each network protocol implements the UPI and provides its own specific routines. The advantage of having all protocols implement the same set of operations is that a common interface is provided for different protocols, thereby simplifying the task of composing a collection of protocols by having a chain of calls of the uniform operations across the protocol stack.

The x-kernel defines two main classes of objects: protocol and session. A protocol object is responsible for creating and maintaining the sessions. A session object is the end point of a channel and each point-to-point channel is terminated at each end by a session. A session object implements the code that interprets messages and maintains the state of the channel. Session objects are dynamically created and deleted as channels are opened and closed. For efficiency, the x-kernel takes the thread-per-message model to avoid a context switch between each protocol level. The whole protocol stack runs as a single process.

In addition to the uniform protocol operations, the x-kernel provides a set of libraries that provides efficient solutions to programming tasks common to all network protocols. Specifically, the support libraries include a message library, a map library, and an event library.

The message library [3] provides a message abstraction that is shared by all protocols in the protocol stack. Messages are passed by reference from one layer to another without copying. Moreover, the copy-free methods are also provided for adding headers and trailers to messages, fragmenting messages into multiple packets and reassembling multiple packets into a single message. The message library is a critical aspect of the x-kernel's performance.

The map library [4] defines a data structure of a map that represents tables of bindings, where each binding is a pair of an external key and an internal id. Typically, an external key is constructed from the message headers, and an internal id is a pointer to a local object. Protocols use the map library to switch messages from low-level protocols to the appropriate session object that handles the messages. Such mappings are defined when network connections are established, and used each time a packet is received from a lower level protocol. The map supports operations for adding new bindings, removing bindings from the set, and mapping one identifier to another.

The event library provides a *timeout* function for protocols. It allows a protocol to specify an event as a procedure to be called at some future time. By registering a procedure with the event manager, protocols can time out and act on those messages that have been not acknowledged.

2.3 The P-Kernel

The p-kernel [7], based on and inspired by the x-kernel, was developed at Carleton University from 2000 onwards for implementing mobile and wireless network protocols applied to the Palm OS. It is implemented in C++ as a library and embedded into an application that requires communications. It inherits most of the design and implementation strategies from the x-kernel. The challenge of the p-kernel is to support PDA operations which are slow and have low memory and few communication resources. In addition, the Palm OS does not support thread and semaphore, which forces the p-kernel to simulate its own [24,25].

Chapter 3

Generative Programming for Telecommunications

Protocols

3.1 Generative Programming

The current object-oriented analysis/design (OOA/OOD) aims to satisfy the requirements for a single system, not a family of systems [13]. The notations used in OOA/OOD might not be able to model the variability without considering the implementation. For example, when a UML class diagram is drawn, it has to be decided whether to use inheritance, aggregation, class parameterization or some others to represent a given variation point.

In order to come up with reusable components, the focus could be moved from single systems to system families. Generative programming (GP) [13-15] is an example of this type of technology that is concerned with modeling families of software systems. In GP, a highly customized and optimized instance of that family can be automatically assembled from the reusable implementation components according to the particular requirements of the system by means of the configuration knowledge.

In GP [13], parameterization is used to represent families of components compactly. The validity of the parameter value combinations is checked because of the dependencies and interactions among them. The components are generated at compile time, thus reducing much of the overhead due to unused code and run-time checks. Configuration knowledge is used to map from the problem space to the solution space. Problems are separated into

distinct sets of code to avoid program code having to deal with many issues simultaneously.

The GP has three main steps: domain analysis, domain design and domain implementation [13,14]. Both domain scoping and feature modeling are involved in domain analysis. Domain scoping determines what is in the domain and what is not. Feature modeling identifies commonality, variability and dependencies within a family of systems. In domain design, a common architecture is developed as a collection of implementation components for the system family, and methods are designed to specify family members and their automatic assembly from elementary components. In domain implementation, the components are implemented.

3.2 A Protocol Implementation Framework for Linux (PIX)

PIX uses the concept of Uniform Interface (UI) and thread-per-message process model from the x-kernel and focuses on a family of telecommunications protocols by using the GP approach [16].

3.2.1 Domain Analysis

In PIX, the Protocol concept is defined to abstractly represent specific communication rules and message formats. The following features of Protocol are provided to model the various properties of the protocol system: Session, UI, ActiveId, ActiveMap, ActiveOpen, Message, Participant and EventManager. The Message feature has a sub-feature called singleBuffMessage.

◆ Protocol

The Protocol concept represents specific communication rules and message formats. It provides communication services for the higher level objects to exchange messages.

◆ Session

The Session feature is the abstraction of an end-point of a communication channel. It implements the code that interprets messages. A session object is dynamically created and deleted when a new channel is opened and closed. Each protocol could create several sessions and the protocol is responsible for the life cycles of sessions.

◆ Uniform Interface (UI)

UI defines the common operations to all protocols and sessions. Each protocol and session has its own specific implementation of UI.

◆ Participant

The Participant feature identifies the participants that communicate with each other.

◆ ActiveOpen

The ActiveOpen feature is required when a node (like a client) knows which other nodes it wishes to communicate with.

◆ ActiveId

The ActiveId feature is a key entry structure containing the identifiers of both the local and the remote participant.

◆ ActiveMap

The ActiveMap feature is an abstraction for mapping bindings between ActiveId and the corresponding active sessions. It also provides operations to maintain a set of bindings: adding, removing and looking up.

◆ Message

A message is an abstraction of the data between the protocol layers and among the networks. It is a string of bytes of some length and exports manipulation operations to the message for the adding and retrieving of headers and data. It could have different implementations according to the system requirements.

◆ EventManager

Scheduling events is a common protocol activity. Some protocols, like TCP, need to perform retransmission whenever data get lost. The retransmission should be implemented as an event that happens after a specified period of time.

The results of the feature analysis can be documented using the feature diagrams shown in Figure 3.1. A feature diagram is a representation that captures the variable features of a concept. The purpose of the feature diagram is to provide hints on the components which are required. The root of the feature diagram represents the Protocol concept. The remaining nodes are features of Protocol. Each node terminates a branch stemming from the root. A feature can be either mandatory (indicated by a filled circle) or optional (indicated by an empty circle). Each feature may become a component category and each sub-feature may become a component. The Protocol concept also becomes a component.

Figure 3.1 Feature diagram of the PIX

In PIX, any two hosts make and maintain the connections with each other by means of the Uniform Interface (UI) provided. There are four main procedures for connection management (shown in Fig. 3.2): active opening of a channel, passive opening of a channel, active closing of a channel from the top of the protocol stack, and passive closing of a channel from the bottom of the protocol stack. In [16], the active open and active close processes are defined while the passive open and passive close processes are left for future work.

Figure 3.2 Process of managing a connection between a client and a server

◆ Active opening of a channel

In computer networks, a local host can issue an active open of a channel to communicate with a remote host when it knows the identifier of the remote host. Corresponding sessions are created and saved in the active map of each protocol layer by layer from the top of the protocol stack to the bottom. A channel is established if the whole process is successful. The identifier here could be the port number and IP address in the TCP/IP suite. Figure 3.3 indicates the process of the active open in the TCP/IP stack. An application program calls a UI operation *Session* xOpen(Participant** participant)* of the TCP protocol to start the active open. This calling process continues to the lowest protocol, the link layer protocol (LLP), where a new LLP session is created and saved in the LLP active map. This session is returned to the IP layer, where an IP session is

created and stored in an IP active map. Eventually, a TCP session is returned to the application program if this back tracking process moves on successfully.

Figure 3.3 Sequence diagram in an active open process

◆ Active closing of a channel from the top of the protocol stack

When an application program is terminated, the high level protocol is told to delete the related session from the active map. This process will trace down to the bottom layer of the protocol stack. The channel will be closed if the process is successful. Figure 3.4 indicates the process of active close in the TCP/IP stack. An application program calls a UI function *Outcome* `xCloseSession(void *aSession)` of the TCP protocol to start an active close. In the TCP layer, the TCP session is erased from the TCP active map. This

process continues to the lowest protocol layer, the link layer, where an *Outcome* value is returned to represent whether the closing process is OK or NOK.

Figure 3.4 Sequence diagram in an active close

3.2.2 Domain Design

In domain design, a GenVoca grammar is constructed to define the dependencies among the components.

- ◆ Identify the main features in the feature diagram from domain analysis

The main features of our protocol are Session, UI, Message, Participant, ActiveOpen, ActiveId, ActiveMap, and EventManager.

- ◆ Express component categories and components per category

The component categories correspond to the main features listed above. The component categories and the components per category for the protocol are shown in Figure 3.5.

Figure 3.5 Component categories for the protocol

- ◆ Identify “use” dependencies between component categories

A components and dependencies diagram identifies the dependencies between the categories. Dependencies are represented by arrows between nodes which correspond to categories or collections of categories.

Figure 3.6(a) depicts a component and dependency diagram for the Protocol concept. UI uses Session, ActiveId, ActiveMap, Participant, Message, and EventManager while ActiveOpen uses UI.

(a)

(b)

(c)

Figure 3.6 Derivation of a layered architecture (a), (b), (c)

- ◆ Sort the categories into a layered architecture

The component categories can be arranged into a hierarchy of layers. Each layer represents a category and the categories that most other categories depend on are moved

towards the bottom of the hierarchy (see Fig.3.6 (b)). Session, Participant, ActiveId, ActiveMap, Message and EventManager do not depend on each other and are called basic categories. For this reason they are collected in a configuration repository, which is used to communicate configuration information to all layers (see Fig. 3.6 (c)). This is possible since a layer may retrieve information from any layer below it. The dashed box around ActiveOpen indicates that this layer is optional.

◆ Write down the GenVoca grammar

The GenVoca grammar of the Protocol concept is defined as shown in Figure 3.7. It is used to formalize the elements of the feature diagram and the dependencies among the categories of components in the protocol family. Categories with the most incoming dependency arrows are placed near the bottom. The bottom layer is the configuration repository. Notice that Configuration Repository is abbreviated to Config.

ActiveOpen	:	ActiveOpen[UI] UI
UI	:	Protocol[Config]
Config:		
Session	:	Session
Message	:	SingleBuffMessage
EventManager:		SimpleEventManager
Participant	:	Participant
ActiveId	:	ActiveId
ActiveMap	:	ActiveMap

Figure 3.7 GenVoca grammar for the protocol family

3.2.3 Domain Implementation

Once the layered architecture has been determined, the components can be implemented.

As stated previously, a component from a given layer takes a component from the layer below it as its parameter, which means that the components are implemented as parameterized components. In C++, class templates are used for this purpose. For example, Protocol is implemented as follows:

Declaration of class Protocol

```
1. template <class Config_>
2. class Protocol
3. {
4.     public:
5.         // publish "Config_" under the name "Config" as a member type
6.         typedef Config_ Config;
7.         // retrieve all the needed types from "Config"
8.         typedef typename Config::Session Session;
9.         typedef typename Config::Message Message;
10.        typedef typename Config::Participant Participant;
11.        // constructor
12.        Protocol();
13.        // destructor
14.        ~Protocol();
15.        // does an active open of a session
16.        // input:
17.        // p : a pair of participant addresses, p[0] is local and p[1] is remote
18.        // output:
19.        // a reference to a session, value NULL reports a failure
20.        virtual Session *xOpen(Participant **p)
21.        { return NULL; }
22.        // does a passive open of a session
23.        // input:
24.        // p : a local participant address, p[0]
25.        // output:
26.        // OK/NOK : succeeded/failed to passive open
27.        virtual Outcome xOpenEnable(Participant **p)
28.        { return OK; }
29.        // undo one passive open of a session
30.        // input:
31.        // p : a participant address, p[0]
32.        // output:
33.        // OK/NOK : succeeded/failed to undo a passive open
34.        virtual Outcome xOpenDisable(Participant **p)
```

```

35.  { return OK; }
36.  // undo all passive open of a session
37.  // input:
38.  // p : a participant address, p[0]
39.  // output:
40.  // OK/NOK : succeeded/failed to undo all passive open related to p[0]
41.  virtual Outcome xOpenDisableAll(Participant **p)
42.  { return OK; }
43.  // called by low level protocol to inform high level protocol that a low level session
44.  // originally opened has been closed by the peer. Therefore, the high level session should
45.  // be closed and erased from the active map
46.  // input:
47.  // lls : a reference to the low-level session
48.  // msg : message
49.  // output:
50.  // OK/NOK : succeeded/failed to demultiplex
51.  virtual Outcome xCloseDone(Message* msg, Session* lls){return OK; }
52.  // demultiplex a message
53.  // input:
54.  // lls : a reference to the low-level session that called the operation
55.  // msg : message being demultiplexed
56.  // output:
57.  // OK/NOK : succeeded/failed to de-multiplex
58.  virtual Outcome xDemux(Session *lls, Message *msg)
59.  { return OK; }
60.  // retrieve information or set the value of parameter of the protocol
61.  // Input:
62.  // opcode : identity of the action
63.  // buffer: pointer to the information or to the value (memory is allocated by the caller)
64.  // length : size of buffer
65.  // output:
66.  // length of information written in buffer, -1 indicates an error
67.  virtual int xControl(int opcode, char *buffer, unsigned int length)
68.  { return 0; }
69. };

```

Protocol takes Config_ as its parameter and publishes it under the new name Config in line 6. Any other component that takes Protocol as its parameter can retrieve Config from it using the C++ scope operator::, i.e., Protocol::Config.

The next important component is ActiveOpen. According to the grammar in Fig. 3.7, it takes a UI as its parameter.

Declaration of class ActiveOpen

```
1.  template <class Base>
2.  class ActiveOpen: public Base
3.  {
4.  public:
5.      // export the name "Base::Config" under the name "Config"
6.      typedef typename Base::Config Config;
7.      // retrieve all the needed types from "Config"
8.      typedef typename Config::Session Session;
9.      typedef typename Config::Message Message;
10.     typedef typename Config::Participant Participant;
11.     typedef typename Config::ActiveId ActiveId;
12.     typedef typename Config::ActiveMap ActiveMap;
13.     static const int hdrSize = Config::hdrSize;
14.     typedef typename Config::Header Header;
15.     typedef typename Config::ProtocolUI ProtocolUI;
16.     // constructor
17.     ActiveOpen();
18.     // destructor
19.     ~ActiveOpen();
20.     // active open of a session. Creates an active session embedded in this protocol. It takes in
21.     // argument a local participant and a remote participant contained in "Participant". When
22.     // the session is created, an entry in the active map is created. The participants are used to
23.     // construct the key. Construction of the key is specific to each protocol.
24.     Session *xOpen(Participant **p)
25.     { ... implementation detail omitted... }
26.     // called by a low level protocol to inform a high level protocol that an opened low level
27.     // session has been closed by the peer. Therefore, the high level session should be closed
28.     // and erased from the active map.
29.     Outcome xCloseDone(Message* msg, Session* lls)
30.     { ... implementation detail omitted... }
31.     // demultiplex a message
32.     Outcome xDemux(Session *lls, Message *msg)
33.     { ...implementation detail omitted... }
34.     // handles the closing of a session
35.     Outcome sessionCloseEvent(void *aSession)
36.     { ...implementation detail omitted... }
37.     protected:
38.     // active map
39.     ActiveMap activeMap;
40.     // create a session and a binding in the active map, has to be
41.     // overloaded by specific protocol classes
42.     virtual Session *createSession(ActiveId *key, ProtocolUPI *p)
43.     { return NULL; }
44.     // extract the header in a message, can be redefined by derived classes
45.     virtual void extractHdr(Message *msg, unsigned int size, char **buff)
46.     {
47.         // precondition
48.         assert(msg && *buff);
49.         *buff = msg->Pop(size);
```

```

50. }
51. // construct a de-multiplexing key
52. virtual void constrDemuxKey(Session *lls, char *buff, ActiveId *key)
53. { }
54. // preamble of operation xOpen(). It has to be overloaded by every specific protocol
55. virtual Outcome xOpenPreamble(ActiveId *key, Participant **p)
56. { return OK; }
57. // tell the high level protocol to delete its related session
58. virtual Outcome xCloseDonePostamble(Message* msg, ActiveId key)
59. { return OK; }
60. // assign a de-multiplexing key from information contained in "s"
61. virtual void assignDemuxKey(Session *s, ActiveId *key)
62. { }
63. // tell the low level protocol to delete its related session
64. virtual Outcome SessionClosePostamble(Session *s)
65. { return OK; }
66. };

```

Here `ActiveOpen` is implemented as an inheritance-based wrapper, that is, a template class derived from its parameters. It overrides the methods in class `Base` and retrieves the configuration repository from its parameters.

In the PIX, a base class for a specific protocol can be constructed by using the type definition, such as `typedef ActiveOpen<Protocol<Config>> Base1`. Different protocols have different component implementations in the configuration repository of `Config`, thereby leading to the various generated base classes. For example, a high level protocol has its specific definitions of `ActiveId` and `ActiveMap`, which are different to those of a low level protocol.

Chapter 4

New Features of PIX

4.1 Overview of New Features

In this thesis, new features and sub features in session management, message modeling and event handling have been added to the Protocol concept in order to improve the functionality of PIX.

◆ PassiveOpen

PassiveOpen, a new feature of the Protocol concept, is required when a node (like a server) is willing to access connections from remote nodes but does not know the origin of the connection request. An Enable object is created to remember the fact that this node was willing to access connections. The willingness to connect with identical participants must have counted references in order to handle the deletion properly. The definition of an Enable structure is:

```
typedef struct enable
{
    // reference count for the calls with identical participants
    int refCnt;
} Enable;
```

◆ PassiveId

The PassiveId feature is a structure of a key entry that only contains the identifiers of the local participant.

◆ PassiveMap

The PassiveMap feature is an abstraction for mapping bindings between PassiveId and the corresponding enabled objects. It also provides operations that maintain a set of bindings: adding, removing and looking up.

◆ BTMessage

BTMessage, a sub feature of the feature Message, provides an alternative buffer tree message model to allocate messages dynamically and avoid data copying.

◆ SimpleEventManager

The SimpleEventManager, a sub feature of EventManager, handles the events when the number of the events is fewer. Each event is attached to a timer and is fired as soon as the timer expires.

◆ DeltalistManager

The DeltalistManager, a sub feature of EventManager, handles the events when several need to be scheduled. A double linked list is created to store the events and an independent thread is created to fire them.

◆ TimingwheelManager

The TimingwheelManager, a sub feature of EventManager, handles the events when several need to be scheduled more efficiently. An array of double linked lists is created to store the events and an independent thread to fire them.

Figure 4.1 depicts an extended feature diagram for the Protocol concept. New features and sub features of PassiveId, PassiveMap, PassiveOpen, BTMessage and EventManager are added to the original one developed in [16]. A feature can have one or several sub features. For example, the feature Message has the sub features SBMessage and

BTMessage, which correspond to two different alternatives for representing messages. The EventManager feature has the sub features SimpleEventManager, DeltalistManager and TimingwheelManager, which are the candidates from which the developer can choose, depending on the specific situation. The alternative features are depicted by an empty arc connecting the edges. Features are subject to constraints that are expressed in additional documentation. For instance, the ActiveOpen feature requires ActiveMap and ActiveId and the PassiveOpen feature requires passiveId and PassiveMap. The diagram in Figure 4.1 shows 24 different protocol variations (two alternative message features, three alternative event features, an optional ActiveOpen, and an optional PassiveOpen, i.e. $2*3*2*2=24$).

Figure 4.1 Extended feature diagram of the PIX

4.2 The PassiveOpen Feature

In the extended PIX, a PassiveOpen feature is added to the Protocol concept to extend the complete session management of the PIX with passive open and close functionality.

◆ Passive opening of a channel

Some hosts (such as a server) are willing to accept communications from other hosts, but do not know in advance which remote host the communication comes from. In this case, the host can only be prepared to accept connections on a specified port and address. In each layer an enable object will be created and stored in the passive map, together with a passive key so that this willingness is remembered. However, the channel is not available yet. Whenever a connection request comes from the remote host, the message will be transmitted upwards and the corresponding sessions will be created layer by layer from bottom to top, using the UI provided by the PIX. If the whole process is successful, a channel is established.

Figure 4.2 shows a passive open process in a TCP/IP stack. An application program calls a UI operation *Outcome* xOpenEnable(Participant** participant)* of the TCP protocol to start the passive open process. A TCP-enabled object is created and stored in the passive map. This calling process continues to the lowest protocol, the link layer protocol (LLP), where an *Outcome* value is returned to signify whether the process is OK or NOK.

Figure 4.2 Sequence diagram of a passive open process

Figure 4.3 gives an example of the session-creating process when a connection request comes from a remote host. The UI operation *Outcome xDemux(Session *lls, Message *msg)* of the link layer protocol is called, in which the active map is checked to see whether a corresponding session exists or not. If the session has been pre-created, an LLP session object is returned to the IP protocol by calling the operation *Outcome xDemux(Session *lls, Message *msg)* of the IP protocol in the UI operation *Outcome xPop(Session *lls, Message *msg, void *hdr)* of the session. If the session does not exist, the passive map is checked to see whether a corresponding enabled object exists. If the enabled object has already been created, an LLP session is created, stored in the active

map and returned to the IP protocol, similarly to the above. This process continues until a TCP session is returned to the application program.

Figure 4.3 Session creating process in a passive open

◆ Passive closing of a channel from the bottom of the protocol stack

When the remote host terminates its application program, a disconnection message will be sent to the local host. The low-level protocol in the local host will be told to erase the related session. This process will trace upwards to the top protocol layer and the channel will be closed if the process is successful. Figure 4.4 demonstrates the passive closing

process in the TCP/IP stack. The UI function *Outcome xCloseDone(Session* Message* msg, Session* lls)* of the LLP protocol starts the passive close process. In the LLP layer, the LLP session is erased from the LLP active map. This process continues on to the application program, where a TCP session is deleted. An *Outcome* value is returned to signify whether the closing process is OK or NOK.

Figure 4.4 Sequence diagram in the passive close process

◆ Write down the GenVoca grammar

The modified GenVoca grammar of the Protocol concept is constructed as shown in Figure 4.5, in which the vertical bars separate alternative components.

PassiveOpen	:	PassiveOpen[ActiveOpen] ActiveOpen
ActiveOpen	:	ActiveOpen[ProtocolUI] ProtocolUI
ProtocolUI	:	Protocol[Config]
Config:		
Session	:	Session
Message	:	SingleBufferMessage TreeBufferMessage
Event Manager:	:	SimpleEventManager DeltalistManager TimingwheelManager
Participant	:	Participant
ActiveId	:	ActiveId
ActiveMap	:	ActiveMap
PassiveId	:	PassiveId
PassiveMap	:	PassiveMap

Figure 4.5 Extended GenVoca grammar for the protocol family

The component to be added is PassiveOpen. According to the grammar in Fig. 4.5, it takes an ActiveOpen as its parameter.

Declaration of class PassiveOpen

```

1. template <class Base>
2. class PassiveOpen: public Base
3. {
4.     public:
5.         // export the name "Base::Config" under the name "Config"
6.         typedef typename Base::Config Config;
7.         // retrieve all the needed types from "Config"
8.         typedef typename Config::Session Session;
9.         typedef typename Config::Message Message;
10.        typedef typename Config::Participant Participant;
11.        typedef typename Config::ActiveId ActiveId;
12.        typedef typename Config::ActiveMap ActiveMap;
13.        typedef typename Config::Enable Enable;
14.        typedef typename Config::PassiveId PassiveId;
15.        typedef typename Config::PassiveMap PassiveMap;
16.        static const int hdrSize = Config::hdrSize;
17.        typedef typename Config::Header Header;
18.        typedef typename Config::ProtocolUPI ProtocolUPI;
19.        // constructor
20.        PassiveOpen(){}

```

```

21. // passive open of a session. Creates an enable object or increase the reference count of the
22. // enable object embedded in this protocol. It takes in argument a local participant
23. // contained in "Participant". When the enable object is created, an entry in the passive
24. // map is created. The participants are used to construct the key. Construction of the key is
25. // specific to each protocol.
26. virtual Outcome xOpenEnable(Participant **p)
27. { ...implementation detail omitted... }
28. // undo the effect of an earlier invocation of xOpenEnable()
29. // i.e., delete the related binding in the passive map
30. virtual Outcome xOpenDisable(Participant** p)
31. { ...implementation detail omitted... }
32. // undo the effect of all earlier invocation of xOpenEnable()
33. virtual Outcome xOpenDisableAll(Participant** p)
34. { ...implementation detail omitted... }
35. // demultiplex a message
36. virtual Outcome xDemux(Session *lls, Message *msg)
37. { ...implementation detail omitted... }
38. protected:
39. // passive map
40. PassiveMap passiveMap;
41. // preamble of operation xOpenEnable(). It has to be overloaded by every specific protocol
42. virtual Outcome xOpenEnablePreamble(PassiveId* key, Participant **p)
43. { return OK; }
44. // preamble of operation xOpenDisable(). It has to be overloaded by every specific
45. // protocol
46. virtual Outcome xOpenDisablePreamble(PassiveId* key, Participant **p)
47. { return OK; }
48. // preamble of operation xOpenDisableAll(). It has to be overloaded by every specific
49. // protocol
50. virtual Outcome xOpenDisableAllPreamble(PassiveId* key, Participant **p)
51. { return OK; }
52. // postamble of operation xOpenEnable(). It has to be overloaded by every specific
53. // protocol
54. virtual Outcome xOpenEnablePostamble(Participant **p)
55. { return OK; }
56. // postamble of operation xOpenDisable(). It has to be overloaded by every specific
57. // protocol
58. virtual Outcome xOpenDisablePostamble(Participant **p)
59. { return OK; }
60. // postamble of operation xOpenDisableAll(). It has to be overloaded by every specific
61. // protocol
62. virtual Outcome xOpenDisableAllPostamble(Participant **p)
63. { return OK; }
64. // construct a passive key from a header buffer
65. virtual void constructPassiveKey(char* buff, PassiveId* key)
66. { }
67. };

```

The `PassiveOpen` is also implemented as an inheritance-based wrapper. The configuration repository is retrieved from `Base` and the methods in class `Base` are overridden.

In the modified PIX, thanks to the alternative and optional features for the Protocol concept, the clients are able to construct the specific bases for their protocol implementations. For example, a client may decide to use `SingleBuffMessage` for a message representation, `DeltalistEventManager` for event handling, and both the passive open and the active open functions for the session management. The type definition will be similar to *`typedef PassiveOpen<ActiveOpen<Protocol<Config>>> Base1`*, while the components of the `SingleBuffMessage` and `DeltalistEventManager` in *`Config`* are selected. Amongst the implementation of the protocols in the protocol stack, a high level protocol may have `ActiveId`, `ActiveMap`, `PassiveId` and `PassiveMap` definitions that are different to a low level protocol.

4.3 The Generator

The feature diagram in Fig. 4.1 defines 24 different protocol configurations. We could define a configuration repository for each of them, but writing configuration repositories is a tedious task. The developers who write them need to know what implementation components are available, what are the valid configurations and which configurations are more optimal. Therefore, it is a burden for application programmers to write these component repositories. Even worse, it means that client code is too closely coupled to the architecture and that changes to the architecture (e.g. the addition of new layers) may

require the modification of all configuration repositories. When there are hundreds or thousands of configurations, writing down each different configuration repository is very difficult.

The solution is to generate configuration repositories from more abstract descriptions. That is exactly what a generator can do for a programmer. A generator takes a system or component specification and returns the finished system or component. The steps [12,13] performed by a configuration generator are shown in Figure 4.6. It first computes defaults to complete the specification, then checks to see if the specified system is valid, and finally assembles the implementation components. In our PIX, there is no need for computing defaults since all the defaults are set through the template parameters. Also, there is no buildability checking since all feature combinations in our family systems are valid.

Figure 4.6 Stages of a configuration generator

C++ has built-in metaprogramming capabilities: template metaprogramming. Because C++ templates constitute a compile-time, turing-complete sub-language of C++, we can use the template instantiation process to perform arbitrary computations at compile time. The generator can take advantage of such C++ properties by using the template for the implementation. For example, it may take the description of a protocol as its parameter

and return the finished protocol in a specially designated member, which is by convention called RET (standing for RETURN). It may also take the description of a session as its parameter and return the finished session, and take the description of a protocol UI as its parameter and return the finished protocol UI for our protocol system.

The following code demonstrates how to create an instance of a ProtocolUI, an instance of a SessionUI, and an instance of a Protocol by using the corresponding generators.

Creation of an instance of template class ProtocolUI

```
1. typedef PROTOCOL_UI_GENERATOR
2.     <SessionUI, Part, BTMessage>::RET ProtocolUI;
```

Creation of an instance of template class SessionUI

```
1. typedef SESSION_UI_GENERATOR
2.     <BTMessage, DeltalistEvent>::RET SessionUI;
```

Creation of an instance of template class Protocol

```
1. typedef PROTOCOL_GENERATOR
2. <
3.     ProtocolUI,
4.     LLPActiveId,
5.     LLPActiveMapConfig::LLPActiveMap,
6.     LLPEnable,
7.     LLPPassiveId,
8.     LLPPassiveMapConfig::LLPPassiveMap,
9.     LLPHeader,
10.    DeltalistEvent,
11.    ProtocolUI,
12.    with_ActiveOpen,
13.    with_PassiveOpen
14. >::RET LLPBase;
```

The implementations of PROTOCOL_UI_GENERATOR, SESSION_UI_GENERATOR and PROTOCOL_GENERATOR are given below. The template structure IF [12] is used to assemble the different components.

Declaration of IF

```
1. template<bool condition, class Then, class Else>
2. struct IF
3. {
4.     typedef Then RET;
5. };
6. // specialization for condition == false
7. template<class Then, class Else>
8. struct IF<false, Then, Else>
9. {
10.     typedef Else RET;
11. };
```

Declaration of a protocolUI generator

```
1. template
2. <
3.     class SessionClass,
4.     // set the default of ParticipantClass to Part
5.     class ParticipantClass = Part,
6.     // set the default of MessageClass to SingleBuffMessage
7.     class MessageClass = SingleBuffMessage
8. >
9. class PROTOCOL_UI_GENERATOR
10. {
11.     // define a short name for the complete generator
12.     typedef PROTOCOL_UI_GENERATOR
13.     <
14.         SessionClass,
15.         ParticipantClass,
16.         MessageClass
17.     > Generator;
18.     // create configuration
19.     struct Config{
20.         typedef SessionClass Session;
21.         typedef ParticipantClass Participant;
22.         typedef MessageClass Message;
23.     };
24.     public:
25.         // assembly components
26.         typedef Protocol<Generator> RET;
27. };
```

Declaration of a session UI generator

```
1. template
2. <
3.     class MessageClass = SingleBuffMessage,
4.     class EventManagerClass = SimpleEvent
5. >
```

```

6. class SESSION_UI_GENERATOR
7. {
8.     // define a short name for the complete generator
9.     typedef SESSION_UI_GENERATOR
10.    <
11.        MessageClass,
12.        EventManagerClass
13.    > Generator;
14. // create configuration
15. struct Config
16. {
17.     typedef MessageClass Message;
18.     typedef EventManagerClass EventManager;
19. };
20. public:
21.     // assembly components
22.     typedef Session<Generator> RET;
23. };

```

Declaration of a protocol generator

```

1. template
2. <
3.     class BaseClass,
4.     class ActiveIdClass,
5.     class ActiveMapClass,
6.     class EnableClass,
7.     class PassiveIdClass,
8.     class PassiveMapClass,
9.     class HeaderClass,
10.    class EventManagerClass,
11.    class ProtocolUIClass,
12.    // set active open functionality to be the default
13.    ActiveOpenFlag activeOpenFlag = with_ActiveOpen,
14.    // set passive open functionality to be the default
15.    PassiveOpenFlag passiveOpenFlag = with_PassiveOpen
16. >
17. class PROTOCOL_GENERATOR
18. {
19.     // define a short name for the complete generator
20.     typedef PROTOCOL_GENERATOR
21.    <
22.        BaseClass,
23.        ActiveIdClass,
24.        ActiveMapClass,
25.        EnableClass,
26.        PassiveIdClass,
27.        PassiveMapClass,
28.        HeaderClass,

```

```

29.     EventManagerClass,
30.     ProtocolUIClass,
31.     ActiveOpenFlag,
32.     PassiveOpenFlag
33. > Generator;
34. // parse domain specific language
35. enum
36. {
37.     isActiveOpen = activeOpenFlag==with_ActiveOpen,
38.     isPassiveOpen = passiveOpenFlag==with_PassiveOpen
39. };
40. // assemble component ActiveOpen
41. typedef IF
42. <
43.     isActiveOpen,
44.     ActiveOpen<Generator>,
45.     BaseClass
46. >::RET ActiveOpenBase;
47. // assemble component PassiveOpen
48. typedef IF
49. <
50.     isPassiveOpen,
51.     PassiveOpen<ActiveOpenBase>,
52.     ActiveOpenBase
53. >::RET ProtocolBase;
54. // create configuration
55. struct Config
56. {
57.     typedef BaseClass Base;
58.     typedef typename Base::Session Session;
59.     typedef typename Base::Participant Participant;
60.     typedef ActiveIdClass ActiveId;
61.     typedef ActiveMapClass ActiveMap;
62.     typedef EnableClass Enable;
63.     typedef PassiveIdClass PassiveId;
64.     typedef PassiveMapClass PassiveMap;
65.     typedef HeaderClass Header;
66.     typedef EventManagerClass EventManager;
67.     typedef typename Base::Message Message;
68.     typedef ProtocolUIClass ProtocolUI;
69. };
70. public:
71.     typedef ProtocolBase RET;
72. };

```

PROTOCOL_GENERATOR evaluates the input parameters, computes types for the configuration repository, wraps ProtocolUI in ActiveOpen (if necessary), wraps

ActiveOpen in PassiveOpen (if necessary), and returns the final protocol base type in RET. The last part of PROTOCOL_GENERATOR is the configuration repository, which we no longer have to implement manually for each protocol configuration. It should be noted that Protocol takes Generator as its parameter rather than Config. Because Config is a member of Generator, Protocol can retrieve Config from Generator. For this reason, we need to modify slightly lines 1 and 7 in the class Protocol, and lines 1 and 4 in the class Session as follows.

Declaration of template class Protocol

```
1. template <class Generator>
2. class Protocol
3. {
4.     public:
5.         // export the name "Generator::Config" under the name "Config"
6.         // makes it accessible to other components
7.         typedef typename Generator::Config Config;
8.         // retrieve all the needed types from "Config"
9.         typedef typename Config::Session Session;
10.        typedef typename Config::Message Message;
11.        typedef typename Config::Participant Participant;
12.        .....
13. };
```

Declaration of template class Session

```
1. template <class Generator>
2. class Session {
3.     public:
4.         typedef typename Generator::Config Config;
5.         typedef typename Config::Message Message;
6.         typedef typename Config::EventManager EventManager;
7.         .....
8. }
```

In conclusion, generators introduce a separation between the feature descriptions in the problem space and the implementation components in the solution space, which allows us

to extend the implementation components by modifying the component structure or adding new components. The existing client code is kept unchanged. This is very useful and convenient for any new features which might be added in the future. Moreover, generators help us to automatically manufacture components or systems from the elementary implementation components.

Chapter 5

Event Feature

5.1 Introduction

In some situations, a protocol that wishes to offer a reliable channel across a network might schedule an event after sending a message, in case the message gets lost. If the protocol does not receive an acknowledgment for the message that was sent within a given period of time, then the protocol fires the event to retransmit the message. Sometimes the protocol may need to broadcast or multicast advertisements or solicitations periodically and scheduling relevant events are required.

G. Varghese [22] introduces seven schemes for managing events. But only two of them are fundamental. The first is a scheme without the storage of events in a data structure and the second is a scheme with the storage of events in a data structure.

With reference to the event handling concepts in [8][22], three sub features – SimpleEventManager, DeltalistManager, and TimingwheelManager – are implemented for the EventManager feature in PIX, with the SimpleEventManager being similar to the first fundamental scheme in [22] and the DeltalistManager and the TimingwheelManager belonging to the second fundamental scheme in [22].

A common event structure is defined as follows:

```
Struct Event {  
    Event *next, *prev;  
    unsigned int deltat;  
    void (*evFn)(void*);  
    void* arg;  
    int state;  
    bool isDetached;  
    pthread_t estub_tid;  
};
```

In the Event structure, two self-pointers, *next* and *prev*, are used to establish the linkage between the nodes when a double linked list is used to store the event record in the DeltalistManager and the TimingwheelManager. The field *deltat* refers to the time period after which the event will be fired. The function pointer *evFn* is set to the function that will be executed after the *deltat*. The pointer *arg* is a parameter passed to the function. The field *state* stores the state of an event. Possible states are E_PENDING (an event is stored in the list), E_SCHEDULED (an event is scheduled), E_RUNNING (an event is being executed), and E_FINISHED (an event has been executed). Memory space allocated for an event can be released when the field *isDetached* is true, but not when it is false. The field *estub_tid* represents a handle to a thread related to the event.

Table 5.1 indicates the operations defined in each event manager, and those operations are explained in detail in the following sections.

Table 5.1 Operations for managing events

Event managers Operations	SimpleEventManager	DeltalistManager	TimingwheelManager
event_stub()	☆	☆	☆
soft_clock()		☆	☆
evSchedule()	☆	☆	☆
evCancel()	☆	☆	☆
evDetach()	☆	☆	☆
fire()		☆	☆
insert()		☆	☆
enqueue()		☆	☆
dequeue()		☆	☆

5.2 The SimpleEventManager Feature

Figure 5.1 shows the event flow in the SimpleEventManager. Whenever an event is created, a new separate thread is created to fire the event. The number of threads corresponds to the number of events.

Figure 5.1 Event flow in the SimpleEventManager

The class SimpleEvent is defined to represent the abstraction of the SimpleEventManager.

Declaration of the class SimpleEvent

```
1. class SimpleEvent
2. {
3.     private:
4.         // a mutex to lock and unlock the access to common data
5.         pthread_mutex_t simple_event_mutex;
6.     public:
7.         // constructor
8.         SimpleEvent();
9.         // destructor
10.        ~SimpleEvent();
11.        // schedule an event
12.        // input:
13.        // evFn: function to be executed
14.        // arg: argument of evFn
15.        // sec: delayed time for firing an event
16.        // output:
17.        // e: return an event object for future management
18.        Event* evSchedule(void(*evFn)(void*), void* arg, unsigned int sec);
19.        // cancel an event
20.        // Input:
21.        // e: an event object to be cancelled
22.        // output:
23.        // state of the canceling operation: EVENT_NOTCANCELLED/EVENT_FINISHED/
24.        // EVENT_RUNNING/EVENT_CANCELLED
25.        EvCancelReturn evCancel(Event* e);
26.        // release space allocated for an event
27.        // input:
28.        // e: an event to be released
29.        void evDetach(Event* e);
30. };
```

The Method *Event* evSchedule(void(*evFn)(void*), void*arg, unsigned int sec)* is used to schedule an event that executes the specified function *evFn* with the given argument *arg* after a delay of *sec* seconds. After an event is created and initialized (lines 5 to 19), a

thread is created to fire the event (lines 27 to 47). At the end of the method, a handle to the event is returned (line 48), which is used to cancel and detach the event.

Method: Event* evSchedule(void(*evFn)(void*), void*arg, unsigned int sec)

```
1. Event* SimpleEvent::evSchedule(void(*evFn)(void*), void*arg, unsigned int sec)
2. {
3.     Event* e;
4.     Event_stubArg* aEvent_stubArg;
5.     // create a new event
6.     if ((e = new(nothrow) Event()) == NULL)
7.     {
8.         aLog.log("SimpleEvent::evSchedule: failed to allocate memory\n");
9.         return NULL;
10.    }
11.    // define the function pointer
12.    e->func = evFn;
13.    // define the argument
14.    e->arg = arg;
15.    // define duration
16.    e->deltat = sec;
17.    e->isDetached = false;
18.    // schedule the event
19.    e->state = E_SCHEDULED;
20.    // create an argument for event_stub()
21.    if ((aEvent_stubArg = new(nothrow) Event_stubArg(e,simple_event_mutex,S_EVENT))
22.        == NULL)
23.    {
24.        aLog.log("SimpleEvent::evSchedule: failed to allocate memory for aEvent_stubArg\n");
25.        return NULL;
26.    }
27.    // create attributes of the received thread
28.    pthread_attr_t attr;
29.    if (pthread_attr_init(&attr) != 0)
30.    {
31.        aLog.log("SimpleEvent::evSchedule(): failed to create thread attribute!");
32.        return NULL;
33.    }
34.    if (pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) != 0)
35.    {
36.        aLog.log("SimpleEvent::evSchedule(): failed to set thread attribute!");
37.        return NULL;
38.    }
39.    // create the receive thread
40.    if (pthread_create(&(e->estub_tid), &attr, &event_stub, (void*)aEvent_stubArg) != 0)
41.    {
42.        aLog.log("SimpleEvent::evSchedule: failed to create thread\n");
43.        delete aEvent_stubArg;
```

```

44.     pthread_attr_destroy(&attr);
45.     return NULL;
46. }
47. pthread_attr_destroy(&attr);
48. return e;
49. }

```

In lines 20 to 26, an argument object *aEvent_stubArg* is created to pass triple parameters to the function *void* event_stub(void* arg)*. A glue class *Event_stubArg* is defined for this purpose.

Declaration of class *Event_stubArg*

```

1. class Event_stubArg
2. {
3.     public:
4.         // an event object
5.         Event* e;
6.         // a thread mutex
7.         pthread_mutex_t pt;
8.         // a flag to tell which event manager is calling
9.         // S_EVENT for SimpleEventManager
10.        // D_EVENT for DeltalistManager
11.        // T_EVENT for TimingwheelManager
12.        int mflag;
13.        // constructor
14.        Event_stubArg(Event* e, pthread_mutex_t pt, int mflag):e(e), pt(pt), mflag(mflag)
15.        {}
16. };
17. void* event_stub(void* arg);

```

The method *void* event_stub(void* arg)*, called in a thread, is where the function associated to the event is executed. For the *SimpleEventManager*, the function will be invoked (line 27) only after the delay of *deltat* seconds is ended (line 20). Lines 21 to 25 set the state of the event to *E_RUNNING* before invoking the function, and lines 33 to 38 set the state of the event to *E_FINISHED*. The memory allocated for the event is released if the field *isDetached* is true. This method and the glue class *Event_stubArg* are

common to the SimpleEventManager, the DeltalistManager, and the TimingwheelManager.

Method: void* event_stub(void* arg)

```
1. void* event_stub(void *arg)
2. {
3.     int old_cancel_state;
4.     // set critical section
5.     pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_cancel_state);
6.     // make the thread asynchronously cancelable
7.     pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
8.     Event_stubArg *aArg = (Event_stubArg*) arg;
9.     Event* e = (Event*) aArg->e;
10.    pthread_mutex_t pmutex = aArg->pt;
11.    int mflag = aArg->mflag;
12.    assert(e->state == E_SCHEDULED);
13.    if(aArg != NULL) delete aArg;
14.    // end of critical section
15.    pthread_setcancelstate(old_cancel_state, NULL);
16.    // check what manager is used
17.    // mflag = S_EVENT, SimpleEventManager
18.    // mflag = D_EVENT, DeltalistManager
19.    // mflag = W_EVENT, TimingwheelManager
20.    if(mflag == S_EVENT) sleep(e->deltat);
21.    // request exclusive access
22.    pthread_mutex_lock( &pmutex );
23.    e->state = E_RUNNING;
24.    // release exclusive access
25.    pthread_mutex_unlock( &pmutex );
26.    // function call
27.    e->func(e->arg);
28.    // if the event is detached
29.    if (e->isDetached == true)
30.    {
31.        delete e; // delete it!
32.    }
33.    else // switch to state E_FINISHED
34.    {
35.        pthread_mutex_lock( &pmutex ); // request exclusive access
36.        e->state = E_FINISHED;
37.        pthread_mutex_unlock( &pmutex ); // release exclusive access
38.    }
39.    return NULL;
40. }
```

Sometimes an event should be canceled. For example, when a protocol receives an acknowledgment from its peer, it needs to cancel the retransmission event. The method *EvCancelReturn evCancel(Event* e)* is used to cancel a given event. The return value is defined by the enumeration type *EvCancelReturn*. It is set to *EVENT_FINISHED* if the event has already happened (lines 10 to 13), *EVENT_CANCELED* if the event has been canceled safely, or *EVENT_NOTCANCELED* if the event cannot be canceled (lines 14 to 23).

Method: EvCancelReturn evCancel(Event* e)

```

1. EvCancelReturn SimpleEvent::evCancel(Event* e)
2. {
3.     // set return value to default
4.     EvCancelReturn res = EVENT_CANCELED;
5.     assert(e);
6.     // request exclusive access
7.     pthread_mutex_lock( &simple_event_mutex );
8.     switch (e->state)
9.     {
10.    case E_FINISHED:
11.        // event is finished
12.        res = EVENT_FINISHED;
13.        break;
14.    case E_RUNNING:
15.    case E_SCHEDULED:
16.        // cancel the thread associated with the event
17.        if (pthread_cancel(e->estub_tid) != 0)
18.        {
19.            aLog.log("SimpleEvent::evCancel: failed to cancel thread\n");
20.            res = EVENT_NOTCANCELED;
21.        }
22.        if(e->isDetached == true) delete e;
23.        break;
24.    default:
25.        // should never enter here
26.        assert( 0 );
27.        break;
28.    }
29.    // release exclusive access
30.    pthread_mutex_unlock( &simple_event_mutex );
31.    return res;
32. }
```

Once the event is completed, the internal resources associated with the event should be freed. The method *void evDetach(Event* e)* releases a handle to an event. The state of the event should be checked. If the event is finished, it can be deleted safely (lines 7 to 9). If the event is to be executed or is being executed, it should not be deleted (lines 10 to 14). In this case, the flag *isDetached* is set to true.

Method: void evDetach(Event* e)

```
1. void SimpleEvent::evDetach(Event* e)
2. {
3.     // request exclusive access
4.     pthread_mutex_lock( &simple_event_mutex );
5.     switch (e->state)
6.     {
7.         case E_FINISHED:
8.             delete e;
9.             break;
10.        case E_RUNNING:
11.        case E_SCHEDULED:
12.            // can not release the handle to the thread associated with the event here
13.            e->isDetached = true;
14.            break;
15.        default:
16.            // should never enter here
17.            assert( 0 );
18.            break;
19.    }
20.    // release exclusive access
21.    pthread_mutex_unlock( &simple_event_mutex );
22.    return;
23. }
```

Note that each event occurs only once. Therefore, if the protocol wants an event to be repeated, the next event should be rescheduled by using *evSchedule()* as the last action taken in the event handling operation. This is true for all event managers.

5.3 The DeltalistManager Feature

If there are many events to be scheduled, the SimpleEventManager creates many threads, which is obviously not efficient. For this reason the DeltalistManager has been developed, using a data structure called *delta list*. The idea is to maintain a doubly linked list of the event records that are sorted in increasing order of the *deltat*. The time stored in each event record is relative to the preceding event. For example, if there are six events scheduled for 1, 2, 5, 5, 8 and 16 seconds in the future, this would result in the delta list shown in Figure 5.2.

Figure 5.2 Delta list implementation of an event manager

Figure 5.3 below depicts the event flow in the DeltalistManager. Events are scheduled and stored in a doubly linked list. A separate thread is created to check the list and to

create other threads to fire the events that only satisfy the fired condition. In this way, the number of concurrently running threads could be reduced.

Figure 5.3 Event flow in the DeltalistManager

The class DeltalistEvent is defined to represent the abstraction of the DeltalistManager.

Declaration of the class DeltalistEvent

```

1. class DeltalistEvent
2. {
3. private:
4.     // a mutex to lock and unlock the access to common data
5.     pthread_mutex_t deltalist_event_mutex;
6.     // first node of a doubly linked list
7.     Event* first;
8.     // A handle to the thread for soft_clock()
9.     pthread_t sc_tid;
10.    // check the linked list and fire events if necessary
11.    static void* soft_clock(void*);
12.    // fire an event
13.    Outcome fire(Event* e);
14.    // put an event after the location pointed by 'after'
15.    void enqueue(Event* e, Event* after);
16.    // get rid of an event from the linked list
17.    void dequeue(Event* e);
18.    // insert an event into the linked list starting with 'head'
19.    void insert(Event* new_event, Event* head);
20.    // create a new thread for soft_clock()
21.    void evInit(int& reval);
  
```

```

22. // disable the standard constructor
23. DeltalistEvent();
24. // disable the standard "copy constructor"
25. DeltalistEvent(const DeltalistEvent &);
26. // disable operator "="
27. const DeltalistEvent& operator=(const DeltalistEvent &);
28. public:
29. // constructor
30. DeltalistEvent(int& reval);
31. // destructor
32. ~DeltalistEvent();
33. // return a handle to a thread
34. pthread_t getSctid(){return sc_tid;}
35. // schedule an event
36. // input:
37. // evFn: function to be executed
38. // arg: argument of evFn
39. // sec: delayed time for firing an event
40. // output:
41. // e: return an event object for future management
42. Event* evSchedule(void(*evFn)(void*), void* arg, unsigned int sec);
43. // cancel an event
44. // Input:
45. // e: an event object to be cancelled
46. // output:
47. // state of the canceling operation: EVENT_NOTCANCELLED/EVENT_FINISHED/
48. // EVENT_RUNNING/EVENT_CANCELLED
49. EvCancelReturn evCancel(Event* e);
50. // release space allocated for an event
51. // input:
52. // e: an event to be released
53. void evDetach(Event* e);
54. };

```

The method *Event* evSchedule(void(*evFn)(void*), void*arg, unsigned int sec)* is used to schedule an event and is defined as follows. It creates an event and incarnates it with the parameters in the same way as the SimpleEventManager does (lines 4 to 15). It then checks the delay time. If the time is equal to zero the event is fired immediately by calling the method *Outcome fire(Event* e)* (lines 25 to 33). However, if the time is greater than 0, the event will be inserted into a doubly linked list by calling the method

void insert(Event, Event*)* (lines 16 to 24). A reference to the event is returned (line 34), which can be used to cancel the event later.

Method: Event* evSchedule (void (*evFn)(void*), void* arg, unsigned int sec)

```

1. Event* DeltalistEvent::evSchedule (void (*evFn)(void*), void* arg, unsigned int sec)
2. {
3.     Event* e;
4.     // create a new event
5.     if ((e = new(nothrow) Event()) == NULL)
6.     {
7.         aLog.log("DeltalistEvent::evSchedule: failed to allocate memory\n");
8.         return NULL;
9.     }
10.    // initialize the event
11.    e->deltat = sec;
12.    e->evFn = evFn;
13.    e->arg = arg;
14.    e->state = E_PENDING;
15.    e->isDetached = false;
16.    if (sec > 0)
17.    {
18.        // insert the event into the double linked list
19.        // request exclusive access
20.        pthread_mutex_lock( &(amp;deltalist_event_mutex) );
21.        insert(e, first);
22.        // release exclusive access
23.        pthread_mutex_unlock( &(amp;deltalist_event_mutex) );
24.    }
25.    // fire the event immediately
26.    else
27.    {
28.        if (fire(e) == NOK)
29.        {
30.            aLog.log("DeltalistEvent::soft_clock: firing event failed.\n");
31.            return NULL;
32.        }
33.    }
34.    return e;
35. }
```

The method *Outcome fire(Event* e)* first sets the state of the event to E_SCHEDULED (lines 4 to 8). A thread is then created to fire the event by calling the operation *void**

event_stub(void arg)* (lines 16 to 36). Note that an argument object *aEvent_stubArg* for *void* event_stub(void* arg)* is created first (lines 9 to 15).

Method: Outcome fire (Event* e)

```

1. Outcome DeltalistEvent::fire (Event* e)
2. {
3.     Event_stubArg* aEvent_stubArg;
4.     // request exclusive access
5.     pthread_mutex_lock( &(deltalist_event_mutex) );
6.     e->state = E_SCHEDULED;
7.     // release exclusive access
8.     pthread_mutex_unlock( &(deltalist_event_mutex) );
9.     // create an argument for event_stub()
10.    if ((aEvent_stubArg=new(nothrow) Event_stubArg(e, deltalist_event_mutex, D_EVENT))
11.        == NULL)
12.    {
13.        aLog.log("DeltalistEvent::fire(): failed to allocate memory for aEvent_stubArg\n");
14.        return NOK;
15.    }
16.    // create attributes of the received thread
17.    pthread_attr_t attr;
18.    if (pthread_attr_init(&attr) != 0)
19.    {
20.        aLog.log("DeltalistEvent::fire(): failed to create thread attribute!");
21.        return NOK;
22.    }
23.    if (pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) != 0)
24.    {
25.        aLog.log("DeltalistEvent::fire(): failed to set thread attribute!");
26.        return NOK;
27.    }
28.    // create the receive thread
29.    if(pthread_create(&(e->estub_tid), &attr, &event_stub, (void*)aEvent_stubArg) != 0)
30.    {
31.        aLog.log("DeltalistEvent::fire(): failed to create thread\n");
32.        delete aEvent_stubArg;
33.        pthread_attr_destroy(&attr);
34.        return NOK;
35.    }
36.    pthread_attr_destroy(&attr);
37.    return OK;
38. }

```

The method `void* soft_clock(void*)` runs in a separate thread that is created as soon as an object of `DeltalistEvent` is created. It checks the linked list and takes actions to fire an event, or not as the case may be. The function contains an infinite loop (lines 6 to 33). In each loop the event manager subtracts one from the *deltat* contained in the first record of the linked list. When the *deltat* value reaches zero, the event and all the subsequent events with the *deltat* value being zero are fired by calling the method *Outcome fire(Event* e)* (lines 16 to 20). The fired events are then dequeued from the linked list (lines 21 to 25).

Method: void* soft_clock (void* arg)

```

1. void* DeltalistEvent::soft_clock (void* arg)
2. {
3.     Event* e;
4.     Event* q;
5.     DeltalistEvent *pThis = (DeltalistEvent *)arg;
6.     while (1) // start of infinite loop
7.     {
8.         q = first;
9.         for(e = q->next; e != q && e->deltat == 0; e = e->next)
10.        {
11.            if(e->func == NULL)
12.            {
13.                aLog.log("DeltalistEvent::soft_clock: e->func == NULL pointer.\n");
14.                continue;
15.            }
16.            if(pThis->fire(e) == NOK)
17.            {
18.                aLog.log("DeltalistEvent::soft_clock: firing event failed.\n");
19.                continue;
20.            }
21.            // request exclusive access
22.            pthread_mutex_lock( &deltalist_event_mutex );
23.            pThis->dequeue(e);
24.            // release exclusive access
25.            pthread_mutex_unlock( &deltalist_event_mutex );
26.        } // end of for loop
27.        // request exclusive access
28.        pthread_mutex_lock( &deltalist_event_mutex );
29.        if(e != q) e->deltat--;
30.        // release exclusive access

```

```

31.  pthread_mutex_unlock( &deltalist_event_mutex );
32.  sleep(1);
33.  }//end of while loop
34.  return NULL;
35. }

```

To schedule a new event, the DeltalistManager moves down the list and inserts the new event record in the appropriate position. It should be noted that the relative time of both the new event and the event immediately following it have to be adjusted. Deleting an event from the list is done in the same way. For example, given the delta list in Figure 5.3, inserting a new event that is to happen 14 seconds in the future results in the new delta list shown in Figure 5.4. Notice that the event record following the new one has been adjusted to reflect the fact that the event that was scheduled for 8 seconds after the fifth event is now scheduled for 2 seconds after the new event.

Figure 5.4 Inserting a new event into a delta list

Three operations are provided to deal with the above issues. The method *void insert(Event* e, Event* head)* inserts the event into a doubly linked list with *head* as the first node. The method *void enqueue(Event* e, Event* after)* executes the process of inserting an event after the node with the event *after*. The method *void dequeue(Event* e)* removes an event from the list and adjust the list accordingly.

Method: void insert(Event* e, Event* head);

```
1. void DeltalistEvent::insert (Event* new_event, Event* head)
2. {
3.     Event* curr;
4.     curr = head->next;
5.     do // start of do-while loop
6.     {
7.         if (curr->deltat <= new_event->deltat)
8.         {
9.             // new_event goes after curr
10.            new_event->deltat - = curr->deltat;
11.        }
12.    else
13.    {
14.        // new_event goes in front of curr
15.        curr->deltat - = new_event->deltat;
16.        enqueue(new_event, curr->prev);
17.        return;
18.    }
19.    curr = curr->next;
20. }while(curr != head); // end of do-while loop
21. // new_event goes at the end
22. enqueue(new_event, head->prev);
23. }
```

Method: void enqueue (Event* e, Event* after)

```
1. void DeltalistEvent::enqueue (Event* e, Event* after)
2. {
3.     e->prev = after;
4.     e->next = after->next;
5.     after->next = e;
6.     e->next->prev = e;
7. }
```

Method: void dequeue (Event* e)

```
1. void DeltalistEvent::dequeue (Event* e)
2. {
3.     e->prev->next = e->next;
4.     e->next->prev = e->prev;
5. }
```

In the `DeltalistManager`, when an event is in a pending state, no thread is created for it. Therefore, in method `EvCancelReturn evCancel(Event* e)`, the manager just cancels the event by dequeuing the event from the doubly linked list and set the field `isDetached` of the event to true (lines 14 to 20). The return value is defined by the enumeration type `EvCancelReturn`. It is set to `EVENT_FINISHED` if the event has already happened (lines 10 to 13), `EVENT_CANCELED` if the event has been canceled safely, or `EVENT_NOTCANCELED` if the event cannot be canceled (lines 21 to 31).

Method: `EvCancelReturn evCancel(Event* e)`

```

1. EvCancelReturn DeltalistEvent::evCancel(Event* e)
2. {
3.     // set return value to default
4.     EvCancelReturn res = EVENT_CANCELED;
5.     assert(e);
6.     // request exclusive access
7.     pthread_mutex_lock( &deltalist_event_mutex );
8.     switch (e->state)
9.     {
10.    case E_FINISHED:
11.        // event is finished
12.        res = EVENT_FINISHED;
13.        break;
14.    case E_PENDING:
15.        //adjust the deltat value right after the dequeued event
16.        e->next->deltat += e->deltat;
17.        // dequeue the event from the list
18.        dequeue(e);
19.        e->isDetached = true;
20.        break;
21.    case E_RUNNING:
22.    case E_SCHEDULED:
23.        // cancel the thread associated with the event
24.        if (pthread_cancel(e->estub_tid) != 0)
25.        {
26.            aLog.log("DeltalistEvent::evCancel: failed to cancel thread\n");
27.            res = EVENT_NOTCANCELED;

```

```

28.     }
29.     // delete e
30.     if(e->isDetached == true) delete e;
31.     break;
32. default:
33.     // should never enter here
34.     assert( 0 );
35.     break;
36. }
37. // release exclusive access
38. pthread_mutex_unlock( &deltalist_event_mutex );
39. return res;
40. }

```

The method *void evDetach(Event* e)* releases a handle to an event. When the event is in the pending state, the event is deleted if the field *isDetached* is true or dequeued from the linked list and deleted if the field *isDetached* is false (lines 7 to 16). The other parts are the same as that in the SimpleEventManager and the explanation is omitted.

Method: void evDetach(Event* e)

```

1. void DeltalistEvent::evDetach(Event* e)
2. {
3.     // request exclusive access
4.     pthread_mutex_lock( &deltalist_event_mutex );
5.     switch (e->state)
6.     {
7.         case E_PENDING:
8.             if(e->isDetached == true)
9.                 delete e;
10.            else
11.            {
12.                e->next->deltat += e->deltat;
13.                dequeue(e);
14.                delete e;
15.            }
16.            break;
17.        case E_FINISHED:
18.            delete e;
19.            break;
20.        case E_RUNNING:

```

```
21.     case E_SCHEDULED:
22.         // can not release the handle to the thread associated with the event here
23.         e->isDetached = true;
24.         break;
25.     default:
26.         // should never enter here
27.         assert( 0 );
28.         break;
29. }
30. // release exclusive access
31. pthread_mutex_unlock( &deltalist_event_mutex );
32. return;
33. }
```

5.4 The TimingwheelManager Feature

The DeltalistManager is good when a large number of events is to be scheduled. But there is a problem. When the linked list becomes longer, the insertion of events can be time consuming because sometimes it is necessary to traverse the entire list in order to find the right place to add an event record. To overcome this problem and retain the efficiency in event management, the simple linear doubly linked list is reorganized into an array of doubly linked lists. We call this kind of management a TimingwheelManager. For example, all the events that are scheduled to take place at one second can be put into the list which starts at the first element in the array; all the events to occur at two seconds could be put into the list which starts at the second element in the array; and so on. The process of scheduling new events will include finding the right element in the array and inserting the event into the list. As time passes, the event manager cycles through the elements of the array. Only the time of the first record in the list corresponding to the

current element needs to be decremented. Fig.5.5 indicates the timing wheel implementation as an array of linked lists.

Figure 5.5 Timing wheel implementation

Figure 5.6 depicts the event flow in the TimingwheelManager. Events are scheduled and stored in an array of doubly linked lists. A separate thread is created to check the array and create other threads to fire the events.

Figure 5.6 TimingwheelManager Event Flow

The class TimingwheelEvent is defined as representing the abstraction of the TimingwheelManager.

Declaration of the class TimingwheelEvent

```
1. class TimingwheelEvent
2. {
3.     private:
4.         // a mutex to lock and unlock the access to common data
5.         pthread_mutex_t timing_wheel_mutex;
6.         // an array of doubly linked lists
7.         Event* wheel[EVENT_WHEEL_SIZE];
8.         // A handle to the thread for soft_clock()
9.         pthread_t sc_tid;
10.        // check the linked list and fire events if necessary
11.        static void* soft_clock (void*);
12.        // fire an event
13.        void fire (Event* e);
14.        // put an event after the location pointed by 'after'
15.        void enqueue (Event* e, Event* after);
16.        // get rid of an event from the linked list
17.        void dequeue (Event* e);
18.        // insert an event into the linked list starting with 'head'
19.        void insert (Event* new_event, Event* head);
20.        // create a new thread for soft_clock()
21.        void evInit (int& reval);
22.        // disable the standard constructor
23.        TimingwheelEvent();
24.        // disable the standard "copy constructor"
25.        TimingwheelEvent(const TimingwheelEvent &);
26.        // disable operator "="
27.        const TimingwheelEvent & operator=(const TimingwheelEvent &);
28.    public:
29.        // constructor
30.        TimingwheelEvent(int& reval);
31.        // destructor
32.        ~TimingwheelEvent();
33.        // return a handle to a thread
34.        pthread_t getSctid(){return sc_tid;}
35.        // Schedule an event
36.        // Input:
37.        // evFn: function to be executed
38.        // arg: argument of evFn
39.        // sec: delayed time for firing an event
40.        // Output:
41.        // e: return an event object for future management
42.        Event* evSchedule(void(*evFn)(void*), void* arg, unsigned int sec);
43.        // Cancel an event
44.        // Input:
45.        // e: an event object to be cancelled
46.        // Output:
47.        // state of the canceling operation: EVENT_NOTCANCELLED/EVENT_FINISHED/
```

```

48. // EVENT_RUNNING/EVENT_CANCELLED
49. EvCancelReturn evCancel(Event* e);
50. // release space allocated for an event
51. // Input:
52. // e: an event to be released
53. void evDetach(Event* e);
54. };

```

The method *Event* evSchedule (void (*evFn)(void*), void* arg, unsigned int sec)* defined below works in a similar way to that in the delta list manager, except that the pending event is stored in a timing wheel (lines 16 to 24).

Method: Event* evSchedule (void (*evFn)(void*), void* arg, unsigned int sec)

```

1. Event* TimingwheelEvent::evSchedule (void (*evFn)(void*), void* arg, unsigned int sec)
2. {
3.     Event* e;
4.     // create a new event
5.     if ((e = new(nothrow) Event()) == NULL)
6.     {
7.         aLog.log("TimingwheelEvent::evSchedule: failed to allocate memory\n");
8.         return NULL;
9.     }
10.    // initialize the event
11.    e->evFn = evFn;
12.    e->arg = arg;
13.    e->deltat = sec;
14.    e->state = E_PENDING;
15.    e->isDetached = false;
16.    if (sec > 0)
17.    {
18.        // insert the event into the timing wheel
19.        // request exclusive access
20.        pthread_mutex_lock( &(timing_wheel_mutex) );
21.        insert(e, wheel[sec % EVENT_WHEEL_SIZE]);
22.        // release exclusive access
23.        pthread_mutex_unlock( &(timing_wheel_mutex) );
24.    }
25.    // fire the event immediately
26.    else
27.    {
36.        if(fire(e) == NOK)

```

```

37.     {
38.         aLog.log("TimingwheelEvent::evSchedule: firing event failed.\n");
39.         return NULL;
40.     }
28. }
29. return e;
30. }

```

The method *Outcome fire(Event* e)* first sets the state of the event to *E_SCHEDULED* (lines 4 to 8). A thread is then created to fire the event by calling the operation *void* event_stub(void* arg)* (lines 16 to 36). Notice that an argument object *aEvent_stubArg* for *void* event_stub(void* arg)* is created first (lines 9 to 15).

Method: Outcome fire (Event* e)

```

1. Outcome TimingwheelEvent::fire (Event* e)
2. {
3.     Event_stubArg* aEvent_stubArg;
4.     // request exclusive access
5.     pthread_mutex_lock( &(amp;timing_wheel_mutex) );
6.     e->state = E_SCHEDULED;
7.     // release exclusive access
8.     pthread_mutex_unlock( &(amp;timing_wheel_mutex) );
9.     // create an argument for event_stub()
10.    if ((aEvent_stubArg=new(nothrow) Event_stubArg(e, timing_wheel_mutex, W_EVENT))
11.        == NULL)
12.    {
13.        aLog.log("TimingwheelEvent::fire(): failed to allocate memory for aEvent_stubArg\n");
14.        return NOK;
15.    }
16.    // create attributes of the received thread
17.    pthread_attr_t attr;
18.    if (pthread_attr_init(&attr) != 0)
19.    {
20.        aLog.log("TimingwheelEvent::fire(): failed to create thread attribute!");
21.        return NOK;
22.    }
23.    if (pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) != 0)
24.    {
25.        aLog.log("TimingwheelEvent::fire(): failed to set thread attribute!");
26.        return NOK;
27.    }

```

```

28. // create the receive thread
29. if(pthread_create(&(e->estub_tid), &attr, &event_stub, (void*)aEvent_stubArg) != 0)
30. {
31.     aLog.log("TimingwheelEvent::fire(): failed to create thread\n");
32.     delete aEvent_stubArg;
33.     pthread_attr_destroy(&attr);
34.     return NOK;
35. }
36. pthread_attr_destroy(&attr);
37. return OK;
38. }

```

In the method *void* soft_clock(void* arg)*, there are three loops in the function: the infinite *while* loop (lines 6 to 36); the outer *for* loop (lines 8 to 34); and the inner *for* loop (lines 11 to 28). The outer *for* loop will go over the array from the smallest element to the largest. The inner *for* loop walks through the linked list of the current element. The event manager subtracts one from the *deltat* contained in the first record of the linked list. When the value of *deltat* becomes zero, the event and all the subsequent events with the *deltat* value being zero are fired by calling *Outcome fire(Event* e)*.

Method: void* soft_clock(void* arg)

```

1. void* TimingwheelEvent::soft_clock(void* arg)
2. {
3.     Event* e;
4.     Event* q;
5.     TimingwheelEvent *pThis = (TimingwheelEvent *)arg;
6.     while (1) // start of infinite loop
7.     {
8.         for (int t = 1; t < EVENT_WHEEL_SIZE; t++)
9.         {
10.            q = pThis->wheel[t];
11.            for(e = q->next; e != q && e->deltat == 0; e = e->next)
12.            {
13.                if(e->func == NULL)
14.                {
15.                    aLog.log("TimingwheelEvent::soft_clock: e->func == NULL pointer.\n");
16.                    continue;
17.                }
18.                if(pThis->fire(e) == NOK)

```



```

19.     {
20.         aLog.log("TimingwheelEvent::soft_clock: firing event failed.\n");
21.         continue;
22.     }
23.     // request exclusive access
24.     pthread_mutex_lock( &timing_wheel_mutex );
25.     pThis->dequeue(e);
26.     // request exclusive access
27.     pthread_mutex_lock( &timing_wheel_mutex );
28. } //end of inner for loop
29. // request exclusive access
30. pthread_mutex_lock( &timing_wheel_mutex );
31. if(e != q) e->deltat--;
32. // release exclusive access
33. pthread_mutex_unlock( &timing_wheel_mutex );
34. } //end of outer for loop
35. sleep(1);
36. } //end of while loop
37. return NULL;
38. }

```

In the `TimingwheelManager`, when an event is in a pending state, no thread is created for it. Therefore, the event is canceled by dequeuing the event from the array of the linked lists and the field `isDetached` of the event is set to true in the method *EvCancelReturn* `evCancel(Event* e)` (lines 14 to 20). The return value is defined by the enumeration type *EvCancelReturn*. It is set to `EVENT_FINISHED` if the event has already happened (lines 10 to 13), `EVENT_CANCELED` if the event has been canceled safely, or `EVENT_NOTCANCELED` if the event cannot be canceled (lines 21 to 30).

Method: `EvCancelReturn evCancel(Event* e)`

```

1. EvCancelReturn TimingwheelEvent::evCancel(Event* e)
2. {
3.     // set return value to default
4.     EvCancelReturn res = EVENT_CANCELED;
5.     assert(e);
6.     // request exclusive access
7.     pthread_mutex_lock( &timingwheel_event_mutex );
8.     switch (e->state)

```

```

9.  {
10. case E_FINISHED:
11.     // event is finished
12.     res = EVENT_FINISHED;
13.     break;
14. case E_PENDING:
15.     //adjust the deltat value right after the dequeued event
16.     e->next->deltat += e->deltat;
17.     // dequeue the event from the list
18.     dequeue(e);
19.     e->isDetached = true;
20.     break;
21. case E_RUNNING:
22. case E_SCHEDULED:
23.     // cancel the thread associated with the event
24.     if (pthread_cancel(e->estub_tid) != 0)
25.     {
26.         aLog.log("TimingwheelEvent::evCancel: failed to cancel thread\n");
27.         res = EVENT_NOTCANCELLED;
28.     }
29.     if(e->isDetached == true) delete e;
30.     break;
31. default:
32.     // should never enter here
33.     assert( 0 );
34.     break;
35. }
36. // release exclusive access
37. pthread_mutex_unlock( &timingwheel_event_mutex );
38. return res;
39. }

```

To schedule a new event, the event manager targets the event to the right element in the array by the *deltat* of the event, and then moves down the list and inserts an event record for the new event in the appropriate position. It should be noted that the relative time of both the new event and the event immediately following the new event must also be adjusted. Figure 5.7 shows the insertion of a new event $B'm$ into an array of lists.

The following routines are the same as that in the DeltalistManager.

```
void evDetach(Event* e);
```

```

void insert(Event* e, Event* head);

void enqueue(Event* e, Event* after);

void dequeue(Event* e);

```

Figure 5.7 Inserting a new event into an array of lists

5.5 Performance Evaluation

The performance of the three event managers can be evaluated with respect to the following aspect: space and latency. The space is the memory required for the events in the event managers and the latency is the time between the invocation of a routine and its completion in an event manager. The space is not compared here since the three event managers share the common structure for the event and use the same space.

Fig. 5.8 shows the performance comparison of the three event managers by evaluating the elapsed time for scheduling/canceling/detaching an event with reference to the number of

the outstanding events. In the test, a number of outstanding events set for times far in the future are scheduled, causing a number of outstanding events. Then one more event set for a time farther in the future than the others is repeatedly scheduled, canceled and detached. The average elapsed time of 10000 iterations is recorded. The results show that the time for the SimpleEventManager and the DeltalistManager increases linearly with the number of outstanding events, whereas the time for the TimingwheelManager has

Figure 5.8. Performance comparison of the three event managers

very small variation with respect to the number of outstanding events. The SimpleEventManager takes more time since it has to create a thread for each outstanding event whereas the DeltalistManager and the TimingwheelManager only need to store the outstanding events in a doubly linked list or an array of the doubly linked lists. Our results are consistent with those in [22].

The SimpleEventManager is simple and straightforward to program. But it takes more time to manage an event and creates lots of threads. It is appropriate when there are a few outstanding events with small values of *deltat* and the time is not critical.

The DeltalistManager uses a doubly linked list to hold the outstanding events and takes less time to manage an event. But the higher the number of the outstanding events, the more time is needed. It is appropriate when there are a few outstanding events and the time is critical. Most of the current BSD kernels use this scheme [22][23].

The TimingwheelManager is based on the DeltalistManager. But instead of a doubly linked list, an array of doubly linked lists is used to hold the outstanding events. It has the least time and very small variations ((0, 0.0401), (200, 0.0412), (400, 0.0424), (600, 0.0415), (800, 0.0431), (1000, 0.0457)) to manage an event and can support many outstanding events. This scheme is used in the x-kernel [8][22] and a modified BSD [23].

Chapter 6

The Message Feature

6.1 Introduction

The message feature is one of the most important features in the Protocol Implementation Framework for Linux (PIX). It provides the data flow across the protocol stack. Conceptually, a message is just a sequence of bytes [1,3,5]. It has data and a specific length. In the PIX, the message feature implementation is optimized for the processing that is encountered in typical network protocols, such as TCP or IP. There are two key characteristics to network message processing. First, headers are appended to the front of messages when they flow out of the protocol stack, and are stripped off the front of messages when they flow into the protocol stack.

Second, the amount of data in messages is often so large that data copying should be avoided as much as possible. The reason is that each time a message is copied from one memory location to another the data is taken from the source location, travels on the computer internal bus, is loaded into the CPU registers and moves along the bus again to the destination location. The time needed is restricted by the bus bandwidth and speed.

6.2 The SingleBuffMessage Feature

The SingleBuffMessage feature [16], similar to the SKBUF messages used in Linux, is implemented as a stack. The advantage of this model is in the simplicity of its implementation. The idea is to represent a message as a stack with the required size. The headers of the protocols and the message data are pushed onto and popped out of the stack by means of the operations: *Push(unsigned int size)*, *Pop(unsigned int size)* and *Peek(unsigned int size)*. *Push(unsigned int size)* is used to set the required *size* of space for a message. *Pop(unsigned int size)* retrieves the *size* of the data from the message stack while *Peek(unsigned int size)* works similarly, except that the data is still kept in the message stack. This simple model is used if the header sizes of all the protocols in the protocol stack can be predetermined when a message is created.

6.3 The BufferTreeMessage Feature

6.3.1 Concepts

Sometimes, the size of a message (including protocol headers and data) is not possibly predetermined. For example, the length of the routing header extension in IPv6 might not be determined when a route is unknown from the source to the destination and a routing algorithm (such as DSR) is needed to find the route.

In this section, we present our implementation of the message feature using a buffer tree model. The key characteristic of this model is that the size of a message allows unknown and the memory for the message can be dynamically allocated.

The BufferTreeMessage feature is inspired by the message model in the x-kernel [3]. Buffers are allocated dynamically. When the size of the message needs to be extended, a new memory buffer, called a LEAF node, is allocated and linked to the existing message by creating a new PAIR node. Thus each space overflow adds two new nodes to the message tree (one PAIR node and one LEAF node). Figure 6.1 shows the message extension process when there is an overflow.

Figure 6.1 Message extension process of an overflow

6.3.2 Data Structures

From the above design idea, the data structure for the buffer tree model is constructed in C++ as follows.

◆ Node

The class Node is defined as the base class for the two derived classes: PairNode and LeafNode, as shown in the class diagram in Figure 6.2.

Figure 6.2 Class diagram of Node, PairNode and LeafNode

The class Node includes two data members. The data member named *nodeType* determines what type of node it is, a PAIR node or a LEAF node. The data member named *refCnt* is a reference count that gives the number of references to this node. A reference count is needed because nodes could be shared among multiple messages.

Declaration of class Node

```

1. class Node
2. {
3.   private:
4.     // node type: LEAF node or PAIR node
5.     int nodeType;
6.     // reference count of node
7.     int refCnt;
8.   public:
9.     // default constructor
10.    Node():nodeType(-1), refCnt(1);
11.    // constructor
12.    // Input:
13.    // nt: define the type of node
14.    // rc: reference count of node
15.    Node(int nt, int rc):nodeType(nt), refCnt(rc);
16.    // destructor
17.    ~Node();
18. };
  
```

◆ LeafNode

A LEAF node contains data in the node. The data member named *size* specifies the length of the buffer in bytes. The data member named *buf* points to the beginning of the memory storing the data.

Declaration of the class LeafNode

```
1. class LeafNode : public Node
2. {
3.     private:
4.         // size of the buffer
5.         unsigned int size;
6.         // space to store the data in the message
7.         char* buf;
8.     public:
9.         // default constructor
10.        LeafNode():Node();
11.        // constructor
12.        // Input:
13.        // s: size for the buffer
14.        // nt: node type
15.        // rc: reference count
16.        LeafNode(unsigned int s, int nt, int rc):Node(nt, rc);
17.        // destructor
18.        ~LeafNode();
19. };
```

A new class named MsgFrag is needed for the definition of class PairNode. It models message data fragments.

◆ MsgFrag

The class MsgFrag represents a message fragment and is the most fundamental structure in a message buffer tree. It refers to a sub area of the data in a message tree. It has a data member named *tree*. It points to the node, which is a sub tree of the message. The data members named *head* and *tail* are indices of the first byte in the sub area and the first byte outside of the sub area respectively. The data length is given by the expression *tail - head*.

Declaration of the class MsgFrag

```
1. class MsgFrag
2. {
3.     private:
4.         // start of a fragment
5.         unsigned int head;
6.         // end of a fragment
7.         unsigned int tail;
8.         // pointer to the node that has the message
9.         Node* tree;
10.    public:
11.        // constructor
12.        MsgFrag();
13.        // destructor
14.        ~MsgFrag();
15. };
```

◆ PairNode

A PAIR node is a pair of fragments. The data member named *l* is the left fragment, which is visited first in a message traversal, and *r* is the right fragment.

Declaration of class PairNode

```
1. class PairNode : public Node
2. {
3.     private:
4.         // left fragment
5.         MsgFrag* l;
6.         // right fragment
7.         MsgFrag* r;
8.     public:
9.         // default constructor
10.        PairNode():Node();
11.        // constructor
12.        // Input:
13.        // loff: start of data in left fragment
14.        // llen: end of data in left fragment
15.        // ln: root of tree in left fragment
16.        // roff: start of data in right fragment
17.        // rlen: end of data in right fragment
18.        // rn: root of tree in right fragment
19.        PairNode(unsigned int loff, unsigned int llen, Node* ln, unsigned int roff,
20.                unsigned int rlen, Node* rn):Node(PAIR_NODE_TYPE, INIT_REF_CNT);
21.        // destructor
```

```

22.     ~PairNode();
23. };

```

The abstraction of a message within the buffer tree implementation is represented as the class `BTMessage`.

◆ `BTMessage`

The main data member is *f*, which is the message fragment containing the data for this message. The data member called *first* represents the first left-most node that is the next one to be written on a header push. There are four data members defined in the class `First`. The data member called *leaf* is a pointer to the left-most leaf node in the message tree. The Data members named *head* and *tail* index the start and end of the data in the leaf respectively. The data member called *isMine* is used to represent ownership. It is `TRUE` if the first leaf is owned by this message. Only the owner has the right to write data into the leaf.

Figure 6.3 shows the class diagram of the `BufferTreeMessage`. `BTMessage` contains `First` and `MsgFrag`, while `MsgFrag` includes `Node` and `First` has `LeafNode`. Both `PairNode` and `LeafNode` are subclasses of `Node`.

Figure 6.3 Class diagram of the `BufferTreeMessage` feature

Declaration of the class First

```
1. class First
2. {
3.     private:
4.         // left-most node
5.         LeafNode* leaf;
6.         start of the data
7.         unsigned int head;
8.         // end of the data
9.         unsigned int tail;
10.        // define the ownership
11.        bool isMine;
12.    public:
13.        // constructor
14.        First();
15.        // destructor
16.        ~First();
17. };
```

Declaration of the class BTMessage

```
1. class BTMessage
2. {
3.     private:
4.         // message fragment
5.         MsgFrag* f;
6.         // pointer to left-most node
7.         First* first;
8.         //helper in destroying message
9.         void msgToughDestroy(Node* n);
10.        // handle the overflow when data are pushed using Push()
11.        char* pushOverflow(unsigned int len);
12.        // handle the underflow when data are popped using Pop()
13.        char* peekUnderflow(unsigned int len);
14.    public:
15.        // constructor
16.        BTMessage(unsigned int len);
17.        // constructor
18.        BTMessage(unsigned int len, void* buf);
19.        // copy constructor
20.        BTMessage(const BTMessage& rhs, int& retval);
21.        // destructor
22.        ~BTMessage();
23.        // destroy message when unused
24.        void Destroy();
25.        // truncate "len" bytes from the end of a message
26.        Outcome Truncate(unsigned int len);
27.        // pull "len" bytes from the end of a message
```

```

28. char* Pull(unsigned int len);
29. // create 'len' size for header
30. char* Push(unsigned int len);
31. // pop 'len' data from the front for header
32. char* Pop(unsigned int len);
33. // peek 'len' data from the front of a message
34. char* Peek(unsigned int len);
35. // join two messages of 'left' and 'right' and form a new message
36. BTMessage* msgJoin(BTMessage* left, BTMessage* right);
37. // break a message into a message fragM with length 'len'
38. void msgBreak(BTMessage* fragM, unsigned int len);
39. };

```

As outgoing messages move down the protocol stack, each protocol attaches its header onto the front of the message. Similarly, as an incoming message moves up the protocol stack, each protocol strips its header from the front of the message. The message objects support the following three methods for pushing, popping and peeking the headers:

char Push(unsigned int len);*

char Pop(unsigned int len);*

char Peek(unsigned int len);*

In the method *char* Push(unsigned int len)*, room for *len* bytes is made from the front of the message. The protocol can then write the header to this space using *void* memcpy(void* des, const void* src, size_t n)*. In the method *char* Pop(unsigned int len)*, *len* bytes of space are retrieved from the front of the message. In method *char* Peek(unsigned int len)*, *len* bytes of space are viewed from the front of the message and the message remains unchanged. Method *void Destroy()* is used to delete a message. The deletion of a message should be dealt with carefully as it is possible that a message buffer is shared by some other messages in the BufferTreeMessage model. A reference count is kept in each message. Only those with a count of zero value can be deleted safely. If the

node is a PAIR node, the sub-trees are visited and all the nodes that are no longer needed are deleted as well.

◆ MsgStackDestroy

An instance of the class MsgStackDestroy is used during the deletion of a message.

Declaration of class MsgStackDestroy

```
1. class MsgStackDestroy
2. {
3.     private:
4.         Node* stackArray[D_STACK_SIZE]; // stack array
5.         int top; // next available location
6.     public:
7.         // constructor
8.         MsgStackDestroy();
9.         // destructor
10.        ~MsgStackDestroy();
11.        // stack operation: push data to the stack
12.        void push(Node* v);
13.        // stack operation: pop data from the stack
14.        Node* pop();
15.        // true if stack is empty
16.        bool isEmpty();
17.        // true if stack is full
18.        bool isFull();
19. };
```

The method *Outcome Truncate(unsigned int len)* removes the *len* bytes from the end of a message. The method *char* Pull(unsigned int len)* reads the *len* bytes at the end of a message.

The BufferTreeMessage supports two methods for manipulating message fragments. The method *void msgBreak(BTMessage* fragM, unsigned int len)* breaks *len* bytes off the front of the original message and places them in the fragment *fragM*. After the operation returns, the original message contains the sequence of bytes that remain after *len* bytes

are removed. The Method *BTMessage* msgJoin(BTMessage* left, BTMessage* right)* attaches *right* to the end of *left* and returns a new message.

Visiting Message Data

The *BufferTreeMessage* provides a method *char* getDataFromMsgTree (BTMessage* m, unsigned int& lenp)* in a class *MsgWalk* for walking the tree and extracting the actual data. It takes the message to be traversed and returns the data as well as the length of the data.

◆ *MsgWalk*

In the class *MsgWalk*, the data member named *f* refers to the fragment to be visited next.

The data member named *stack* keeps track of the fragments to be visited.

Declaration of the class *MsgWalk*

```
1. class MsgWalk
2. {
3.     private:
4.         MsgFrag* f;
5.         MessageFragStack* stack;
6.         // disable the standard constructor
7.         MsgWalk();
8.     public:
9.         // constructor
10.        MsgWalk(int& reval);
11.        // destructor
12.        ~MsgWalk();
13.        // the next fragment to be visited
14.        void nextFragment(MsgFrag* f);
15.        // initialize the context w with a message
16.        void msgWalkInit(BTMessage* m);
17.        // helper to visit the message tree
18.        char* msgWalkToughNext(unsigned int& lenp);
19.        // walk along the buffer tree and return a segment of message data
20.        char* msgWalkNext(unsigned int& lenp);
21.        // walk along the buffer tree and return all message data and length
40.        // Input:
41.        // m: message tree to be traversed
42.        // lenp: reference to length of data in a message
```



```

43. // Output:
44. // return a pointer to the beginning of a message
45. char* getDataFromMsgTree(BTMessage* m, unsigned int& lenp);
22. };

```

◆ MessageFragStack

An instance of class MessageFragStack is used to facilitate retrieving the data from the message tree. The stack is initialized with the fragment in the message to be visited. The fragment at the top of the stack is the one to be visited next.

Declaration of class MessageFragStack

```

23. class MessageFragStack
24. {
25.     private:
26.         MsgFrag* stackArray[F_STACK_SIZE]; // stack array
27.         int top; // next available location
28.     public:
29.         // constructor
30.         MessageFragStack();
31.         // destructor
32.         ~MessageFragStack();
33.         // push next visited message fragment to stack
34.         void pushFragment(MsgFrag* mf);
35.         // pop next visited message fragment from the stack
36.         MsgFrag* popFragment();
37.         // true if stack is empty
38.         bool isEmpty();
39.         // true if stack is full
40.         bool isFull();
41. };

```

6.4 Performance Evaluation

Table 6.1 shows the performance evaluation [20,21] of the SingleBuffMessage feature and the BufferTreeMessage feature in the PIX. In the test program, two layers of

protocols, a high level protocol and a low level protocol, are used to simulate the process of establishing a channel, sending and receiving messages. The tests are executed on a machine running Linux Mandrake OS. The sending and receiving processes are repeated 1000 times and the average time is recorded.

1. Since only the pointer to the message is transmitted among the protocol stack, the size of a message makes no difference to the time needed for sending and receiving a message among the protocol stack.
2. It takes longer time to receive a message than to send one. This is because more routines are executed in the receiving process.
3. Sending and receiving a message with the BufferTreeMessage feature takes more time than sending and receiving a message with the SingleBuffMessage feature. Therefore, whenever the size of messages can be predetermined, the SingleBuffMessage feature should be preferred. But if the size of messages is not predictable when the message is created, the BufferTreeMessage feature should be selected, even though it is less efficient than the SingleBuffMessage feature.

Table 6.1 Performance evaluations

	SingleBuffMessage		BufferTreeMessage	
	Time (ms)		Time (ms)	
	Sending	Receiving	Sending	Receiving
1 bytes	0.0036	0.0073	0.0192	0.0356
10 bytes	0.0036	0.0072	0.0193	0.0380
100 bytes	0.0037	0.0073	0.0201	0.0369
1000 bytes	0.0037	0.0075	0.0251	0.0422

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Generative Programming (GP) is a software engineering approach that models a family of software systems. In GP, concepts are defined to represent any elements that have variation points. Features model the variability of a concept. Developed using GP, the protocol implementation framework for Linux (PIX) provides an experimental environment for protocol implementation in mobile and wireless communications and has better configurability. In PIX, the Protocol concept is defined to represent an abstraction of the telecommunications rules. The ActiveOpen features of the Protocol concept for the creation, management and release of a connection provide PIX with functionality for clients. The SingleBuffMessage feature models a pre-allocated message.

In this thesis, several implementation components are developed. The PassiveOpen feature improves the PIX with a passive open functionality, which is required by a server node. Three generators - a protocol generator, a UI generator and a session generator - allow developers to configure the system as required and conveniently add more functionality. The BufferTreeMessage feature has the ability to dynamically allocate memory for a message for the situation where the size of a message is not predictable, while the data copying is avoided. Three event features, the SimpleEventManager, the DeltalistManager and the TimingwheelManager, are implemented for event handling in

PIX. The SimpleEventManager takes more time to schedule new events and could be used when few events exist and the time is not critical. The DeltalistManager and TimingwheelManager are recommended for handling many events, in which the TimingwheelManager is more efficient in scheduling new events than the DeltalistManager.

PIX provides 24 possible configuration repositories and allows the developers to make a selection and manufacture a specific protocol system to satisfy their requirements. A protocol generator, a UI generator, and a session generator are used to assemble the implementation components automatically.

A research project on telecommunications among LEO satellites and ground stations is being developed, using the PIX as an experimental environment. Based on the fundamental functionality provided by the PIX and the hooked protocols, such as SNMP, FTP, STP, DSR, IPv6, and a wireless LAN driver, the communications among the LEO satellites and the ground stations are expected to be realized.

7.2 Future Work

The current PIX supports asynchronous protocols, which means that the protocols do not block to wait for a reply from their peer. But some protocols, such as SNMP, might need to be synchronous. That is, the caller blocks until a reply from the callee is returned. Obviously, the xPush/xDemux/xPop scheme provided in the current PIX does not work well for synchronous protocols since it is not coded to handle the blocking and the unblocking. Therefore, a new set of operations for sending and receiving messages

should be implemented to accommodate synchronous protocols. Inspired from the x-kernel [5], the uniform interfaces could be like:

Outcome xCall(Message request, Message* reply);*

Outcome xCallPop(Message request, Message* reply, void* hdr);*

Outcome xCallDemux(Session s, Message* request, Message* reply);*

The argument *request* corresponds to the given message and the argument *reply* is the replied message. Those operations are synchronous, meaning that each one blocks and does not return until the corresponding reply message is available. The blocking and unblocking of a process could be realized by using the conventional counting semaphores.

Bibliography

- [1] 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
- [2] L. L. Peterson and et al, *The x-kernel: A Platform for Accessing Internet Resources*, IEEE Computer, 23(5): 23-33, May 1990
- [3] D. Mosberger, *Message Library Design Notes*, Network System Research Group, Department of Computer Science, University of Arizona, January 1996.
- [4] D. Mosberger, *Map Library Design Notes*, Network System Research Group, Department of Computer Science, University of Arizona, Jan. 1996.
- [5] Network System Research Group, Department of Computer Science, University of Arizona, *x-kernel Programmer's Manual (Version 3.3)*, January 1996.
- [6] L.L. Peterson, *Getting Started with the x-kernel*, Network System Research Group, Department of Computer Science, University of Arizona, January 1996.
- [7] S. Robinson and M. Barbeau, *The P-kernel: A Framework for Mobile and Wireless Network Protocol Implementation on Personal Digital Assistants*, <http://scs.carleton.ca/~barbeau>, 2000.
- [8] L.L. Peterson and B.S. Davie, *Computer Networks: A System Approach*, Morgan Kaufmann Publishers, San Fransico, CA, First Edition, 1996.
- [9] L.L. Peterson and B.S. Davie, *Computer Networks: A System Approach*, Morgan Kaufmann Publishers, San Fransico, CA, Second Edition, 2000.
- [10] A. S. Tanenbaum, *Computer Networks*, Third edition, Prentice Hall PTR, NJ 1996
- [11] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP*, Vol. II, Second edition, Prentice Hall, NJ, 1994
- [12] G. R. Wright and W. R. Stevens, *The Implementation - TCP/IP Illustrated*, Vol. 2, Addison Wesley, 1999.
- [13] K. Czarnecki and U. W. Eisenecker, *Generative Programming – Methods, Tools, and Applications*, Addison Wesley, 1999.

- [14] K. Czarnecki and U. W. Eisenecker, *Components and Generative Programming*, Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Springer-Verlag, 1999.
- [15] K. Czarnecki and et al, *Generative Programming and Active Libraries (Extended Abstract)*, <http://www.extreme.indiana.edu/~tveldhui/papers>, 1998.
- [16] M. Barbeau, *Generative Programming for Telecommunications Protocols*, <http://scs.carleton.ca/~barbeau>, 2001.
- [17] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, Fifth edition, AddisonWesley Inc., 1998
- [18] Robert Switzer, *Operating Systems - A Practical Approach*, Prentice Hall International (UK) Ltd., 1993
- [19] C.X. Guo and S. Zheng, *Analysis and Evaluation of the TCP/IP Protocol Stack of LINUX*, IEEE Communication Technology Proceedings, WCC-ICCT 2000, International Conference On, Vol.1, 2000, page(s): 444-453
- [20] W. R. Stevens, *UNIX Network Programming*. Prentice Hall PTR, Upper Saddle River, NJ, 1998
- [21] M. Mitchell, J. Oldham and A. Samuel, *Advanced Linux Programming*, New Riders, 2001.
- [22] G. Varghese and A. Lauck, *Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility*, IEE/ACM Transactions on Networking, Vol.5, No.6, page(s): 824-834, December 1997
- [23] A.M. Costello and G. Varghese, *Redesigning the BSD Timer Facilities*, Software – Practice and Experience, Vol. 28(8), page(s): 883-896, July 1998
- [24] Z.-S. Zeng, *An Experimental Implementation of Mobile IPv6 on a Personal Digital Assistant*, Master Thesis, School of Computer Science, Carleton University, 2001.
- [25] Z. Zeng and M. Barbeau, *An Experimental Implementation of Mobile Ipv6 on Personal Digital Assistants*, <http://scs.carleton.ca/~barbeau>, 2000.
- [26] J. Case et al, *A Simple Network Management Protocol (SNMP)*, Internet Request for Comments RFC 1157, May 1990.

- [27] J. Postel and J. Reynolds, *File Transfer Protocol (FTP)*, Internet Request for Comments RFC 765, Oct. 1985.
- [28] Tom Henderson, *Documentation for Satellite Transport Protocol (STP)*, http://www.tomh.org/stp/stp_doc.pdf, Aug. 2000
- [29] S. Deering and R. Hinden, *Internet Protocol, Version 6 (Ipv6) Specification*, Internet Request for Comments RFC 2460, May 1998.
- [30] D. B. Johnson et al, *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks*, Internet-Draft, March 2001.