Study on Transparently Extending Networking Services Framework for Adaptive Communications

ネットワーク機能を透過的に拡張する適応型通信のためのフレームワークの研究

Ryota Kawashima
川島 龍太

Dissertation

January, 2010

Submitted in the fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Informatics,
School of Multidisciplinary Sciences,
The Graduate University for Advanced Studies (SOKENDAI),
Japan

abstract

Since current networking technology is rapidly evolving, many users can now access information from all over the world without any geographical and temporal constraints. Networks are now a fundamental infrastructure for social services, such as E-commerce, online banking, social networking services (SNS), and cloud computing. As a result, current networks must provide a flexible service composition and robust system components.

Most networks are currently based on TCP/IP protocol suite, and any application that supports it can communicate with another application without custom-made protocols. Networks have evolved by incorporating extended functions on the TCP/IP core functions that are complementary without actually changing them. Current networks basically have both old-fashioned systems, and cutting-edge systems.

There is a hierarchical structure referred to as an hourglass based on TCP/IP in the background of the universality of the fundamental networking capability. The main idea of this structure is that the TCP/IP ensures there is a logical connectivity between end-to-end applications. Users can introduce a variety of applied networking services on TCP/IP and physical access technologies under TCP/IP. However, since there still are many systems that only support traditional networking services, it is difficult to widely deploy advanced services in the network, and users may struggle to replace core network protocols with new protocols to correct a backward compatibility problem.

Generally, it is believed that networking applications and services are evolving as the network environment expands and diversifies. However, a variety of dedicated systems that depend on particular platforms and the developing environment have appeared that lack a higher-level common platform for these advanced networking services and users have developed their own services, respectively.

In this paper, we would like to discuss a common software platform for diversifying networking services. Not the only network software, such as the applications, servers, and

protocol stacks, but also the methodologies and implementation techniques are expected to provide a common software platform. In particular, instead of developing brand-new network systems, existing software assets and technologies are used to construct the advanced network environment.

To achieve extensible network softwares, we have focused on a notion of an adaptive communication. The adative communication provides a way to adaptively compose networking capabilities of applications when the surrounding environment is changed. In practice, each networking function for the application could be added, removed, or replaced per functional unit, and therefore, a transparent mechanism that dynamically composes networking functions is required to ensure compatibility with existing systems.

Network applications usually use socket interfaces that abstract the networking functions to communicate over the network. A socket interface generally supports two types of communication: connection-oriented communications and connectionless-oriented communications, and it allows for the construction of network software that is independent of the network protocol. So, I am proposing a system that enables networking functions to be transparently change an application by using the abstraction of the socket interface. Since the change in networking functions is conducted entirely within the socket interface, the application can use the socket interface as is.

Many related systems that transparently change the networking functions of applications. However, there are some platform, performance, developer-oriented behavior, usability, and flexible functions composition restrictions that exist with these systems. This study aims at a more pragmatic system compared to these systems in that it has a component-oriented function, composition mechanism, minimum overhead, multi-platform support, and a unified usability by taking into account not only the methodology for transparent extension, but also a common-platform for higher-level services characteristic. In addition, we take into account a user-interface that targets non-professional users as well as developers in the paper.

In this paper, the proposed system was designed and implemented to work on Windows

and Linux operating systems, and evaluated its functionalities comparing with existing systems. The result showed following advantages;

- Users can extend and configure their systems as intended
- Non-experimental users also can utilize the proposed system without programming
- The proposed system can be available with more applications and operating systems
- Users can manage the proposed system without difficulty

Therefore, the proposed system allows us to extend applications more concisely and finely on variety of network environments, and existing software asset can be leveraged to compose advanced network environment.

In addition, performance overhead of the proposed system was measured by comparing with another application that equips network functions directly. The result showed that overhead of transmitting/receiving each user data was slightly several thousand clocks, and this value can be negligible considering that fluctuation time of the packet processing within the kernel protocol stack and packet transferring on the actual network. As a result, the proposed system could not be performance bottleneck on practical network systems.

論文要旨

近年，ネットワーク技術が急速に発展するに連れて，多くのユーザが地理的，時間的制約を超えて世界中の情報にアクセスできるようになった．また，高速で安全，そしていつでもどこでも通信を利用できる環境が整っているため，ネットワークは単なるリモート間接続のための媒体ではなく，E コマース，インターネットバンキング，ソーシャル・ネットワーキング・サービス (SNS)，クラウド・コンピューティングなどのより高次のサービスを提供するための社会インフラとして利用されている．そのため，現代ネットワークには利用者の多様な要求に答えられる柔軟性とシステムの頑強性が広く求められている．

現在では，多くのネットワークが TCP/IP にその基礎を置いており，TCP/IP に対応したアプリケーションであれば，これらのネットワークに接続して通信を行うことができる．TCP/IP は登場以来ほとんど基本機能が変化をしておらず，ネットワーク機能の拡張は TCP/IP を補完するという形で行われてきた．そのため，従来の機能のままにネットワークを利用しているアプリケーションが数多く存在している一方で，ネットワーク技術の進展に伴い，先進的な機能を実現しているアプリケーションも共存しているというのが現代ネットワークの実情になっている．

このような現代ネットワークが持つ基本機能の普遍性の背景として，TCP/IP を中心とした Hourglass に例えられる機能階層構造が挙げられる．エンドノード間での通信接続性をスタンダード技術である IP および TCP が保証することにより，ユーザは多様なサービス機能や通信技術を導入することが可能になる．しかしながら，先進的なネットワーク機能やサービス機能が相次いで登場するため，それらに対応したシステムとそうでないシステムがネットワーク内に共存することになり，普遍性を持つ高次のサービスを展開できないという問題がある．さらには，後方互換性のために，いったん普及してしまった中核技術の抜本的な転換を行うことが困難になるという懸念もある．

一般に，ネットワーク環境が多様化・高度化するにつれて，それを利用するシステムやサービスも高度化していくと考えられている．しかし，このような機能を利用するための高いレベルでの共通基盤となる仕組みが存在しないため，各利用者がそれぞれの独自の

サービスを提供した結果，プラットフォームや開発言語などの特定環境に依存したシステムが数多く誕生する結果となった．

そこで本研究では，ネットワークを構成するアプリケーションプログラムやサーバプログラム，さらにはオペレーティングシステム内のプロトコルスタックなどのネットワークソフトウェアに焦点を当て，ネットワークの拡張性や後方互換性に配慮したソフトウェアシステムの実現について，その方法論や実装手法などの考察を行い，多様化するサービス機能を実行するための共通基盤となるソフトウェアシステムの実現を目指す．特に，全く新規のネットワークシステムの実現を目指すのではなく，既存の枠組みの中で，多くのネットワークソフトウェアに拡張性を持たせることによって，先進的なネットワーク機能やサービスを幅広く利用できるようなネットワークシステムの実現を目標とする．

拡張性を有するネットワークソフトウェアを実現するため，筆者らは適応型通信という概念に着目した．適応型通信では，ネットワークの機能性や通信環境の変化に適応してアプリケーションのネットワーク機能を構成することができる．具体的には，アプリケーションのネットワーク機能を機能単位で追加／削除／置換する必要があるが，既存システムとの親和性を考慮すると，直接的なシステム変更を行わずにネットワーク機能を構成するための仕組みが必要になる．

ネットワーク機能を利用するアプリケーションは，通常，ネットワーク機能を抽象化したソケットインタフェースと呼ばれる API を利用する．ソケットインタフェースには基本的にコネクション型通信，コネクションレス型通信の二種類が提供されており，個々のプロトコルの詳細には依存しないネットワークソフトウェアの構築が可能になる．そこで本研究では，ソケットインタフェースが持つ抽象性という特徴に着目し，アプリケーションが利用するネットワーク機能をアプリケーションからは見えない形で変更可能にする．このネットワーク機能の変更はソケットインタフェースの内部で行われるため，既存のアプリケーションや OS に対して透過性がある．提案方式を用いることで，後方互換性を保ちつつネットワーク環境に応じてアプリケーションの機能構成を変更できる適応型通信を実現することが可能になる．

これまでにも，本研究で提案するシステムと同様の役割を果たすシステムが数多く提案されてきた．しかしながら，これらの既存システムには，プラットフォーム依存性，多

大な性能オーバヘッド，開発者指向のユーザビリティ，柔軟性に欠けた機能構成などといった問題があった．本研究では，従来研究のように透過的な機能追加を実現するための方法論だけではなく，高度サービスのための共通基盤という性質に配慮し，コンポーネント指向に基づいたネットワーク機能の可結合性，低オーバヘッド，マルチプラットフォーム対応，統一的な操作性などの性質を満たすシステムの実現を目指す．また，開発者だけではなく，専門的知識を持たない利用者を対象としたユーザインタフェースについての考察も行う．

本研究では，Windows および Linux 上で動作するように提案システムの設計・実装を行った．そこで，提案システムの機能性を既存システムと比較した結果，

- ユーザの思い通りに既存システムを拡張できる
- 専門的な知識が無くても機能拡張を行うことができる
- より多くのアプリケーションや OS に対して機能拡張を行うことができる
- 運用面での負荷が小さい

という利点があることが明らかになった．提案システムを用いることで，ユーザは多様なネットワーク環境下で，より手軽に，より詳細に機能拡張を行えるようになるため，既存のソフトウェア資産を活用して先進的なネットワーク環境を構築することができる．

また，提案方式による性能オーバヘッドを評価した結果，１回のデータ送受信に伴うオーバヘッドは数千クロック程度であり，これはカーネル・プロトコルスタック内でのパケット処理時間や実ネットワーク上での転送時間の揺らぎと比較すると極めて小さい値である．したがって，提案システムを実システム環境に導入しても性能上のボトルネックにはならないと考えられる．

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Networks have become social infrastructures that are still evolving. Many people have innovated advanced capabilities and services for traditional TCP/IP networks. However, this diversity of networks makes is difficult for users to yield flexible and environment-adaptable network systems under a unified technology.

In this chapter, the current status and problems of networks are described as a background for this study. Next, several practical methodologies including the current cutting-edge studies are discussed for solving these problems, and then our proposed system is introduced with its advantages over the existing systems. Finally, the organization of this paper is presented.

1.1 Present and Issues of Network

Since current networking technology is rapidly evolving, many users can access information from all over the world without any geographical and temporal constraints. Currently, networks have become a fundamental infrastructure for social services such as E-commerce, online banking, social networking service (SNS), and cloud computing. Moreover, the idea of a ubiquitous network[1] has attracted an enormous amount of interest to bring innovation to networks based on our own individual lifestyles. For instance, people can access networks with their familiar devices to make their life more convenient. Such higher-level services have to be provided on efficient and flexible network systems.

Most current networks are based on the TCP/IP protocol suite, and any applica-

tion that supports it can communicate with another application without custom-made protocols. However, because the TCP/IP protocol suite provides only a fundamental connectivity and typical networking services[1] between end-to-end applications, additional protocols are required for practical applications. In practice, networks have evolved by incorporating extended functions on the TCP/IP core functions complementary without actually changing them. This implies that current networks simultaneously contain both old-fashioned and cutting-edge systems.

There is a hierarchical structure referred to as a hourglass based on TCP/IP (See Fig. 1.1) in the background of the universality of fundamental networking capability. The main idea of this structure is that TCP/IP ensures there is a logical connectivity between end-to-end applications. Users can introduce a variety of applied networking services on TCP/IP and physical access technologies under TCP/IP. However, since many systems that only support traditional networking services, it is difficult to widely deploy advanced services in the network, and users may struggle to replace core network protocols with new protocols for backward compatibility problems. As a result, users will continue to use traditional protocols by exploiting system loopholes (a NAT system is an actual example of this problem in that it is originally introduced as a temporary solution to cover an IPv4 global address space).

Network applications and networking services are evolving as the network environment expands and diversifies, but a variety of dedicated systems depending on a particular platform or the development environment have been developed because of the lack of a higher-level unified platform for advanced networking services. As a result, advanced networking services have to be separately developed for many environments without a unified system interface.

In practice, current network applications need to be able to support upper-layer services[2], such as security and QoS functions. In addition, rapidly growing mobile applications also have to also have to be able to support mobility services like location-awareness

[1]In this paper, a networking service refers to a logical function composed of a set of computation or communication procedures, and one service mainly corresponds to one network protocol.

[2]In this paper, upper-layers refers to session-layers, presentation-layers, and application-layers.

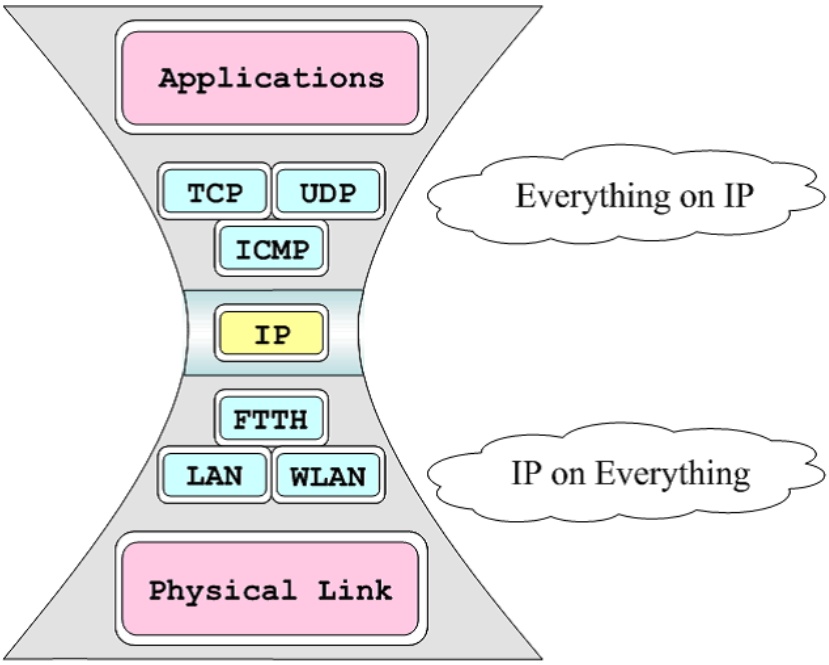

Figure 1.1: The IP hourglass structure. Applied services are put on IP protocol and access protocols are located under IP protocol.

and session migration functions. These services often require some expertise for developing and operating them. As a result, incorporating advanced services into existing user systems tends to be an experimental work, and this requires too many system modifications. Some services, such as those for security purposes, require continuous service upgrades, which can burden users with operations for their systems. Therefore, a scheme for easily introducing and experimenting on given services is required for current network systems.

The concept of *autonomic computing*[2] has recently gathered a lot of attention for more easily managing computer systems with higher-level human guidance. With an autonomic computing environment, users (administrators) are able to effortlessly integrate a new component into existing systems. Therefore, this idea could be a key solution to creating highly-diversified network software that can adapt to a given network environment.

1.2 Adaptation of Networking Capability in Software

In the previous section, the structural problem in current network environments was referred to as rigidly designed and implemented of network software. To keep up with the ongoing growth of network environments, network softwares should have the flexibility mechanism to fit with the surrounding network environments. In this study, we focus on flexible and composable networking functions that can be used by many existing network systems. That is, networking service implementations can be *added to/removed from* application structures flexibly without any modification of the existing system. If such service implementations are available for many existing applications on various platforms, old software assets will not have to be discarded.

The notion of *software adaptation*[3][4] could be a key technology making network software more flexible. Software adaptation has been defined as *enabling the software to modify its structure and behavior dynamically in response to environmental changes.* According to Philip et al.[5], there are two general approaches to implement software adaptation – *Parameter adaptation* and *Compositional adaptation.*

Parameter adaptation has been used to alter the behavior of an application by adaptively calibrating the necessary parameters in accordance with the environmental changes. In most network systems, the TCP protocol uses the parameter adaptation mechanism such that TCP adjusts the transmission rate based on the network congestion or amount of available buffer amounts. In particular, a window size parameter is used to determine a balance between the efficiency and fairness of the communication.

Figure 1.2 shows a conceptual diagram for parameter adaptation. The application's core module periodically checks the surrounding environments, and it updates the parameter settings when the environment is changed. Since these parameters are controlled by the application under a certain strategy, each parameter can be updated by the application itself. Due to this simplicity, this adaptation mechanism has already been implemented in a lot of network software. However, this mechanism has two big drawbacks in that the application's behavior cannot be changed drastically like a full-replacement of the algo-

rithms or protocols, and the application developers cannot incorporate future functional requirements into the application during development.

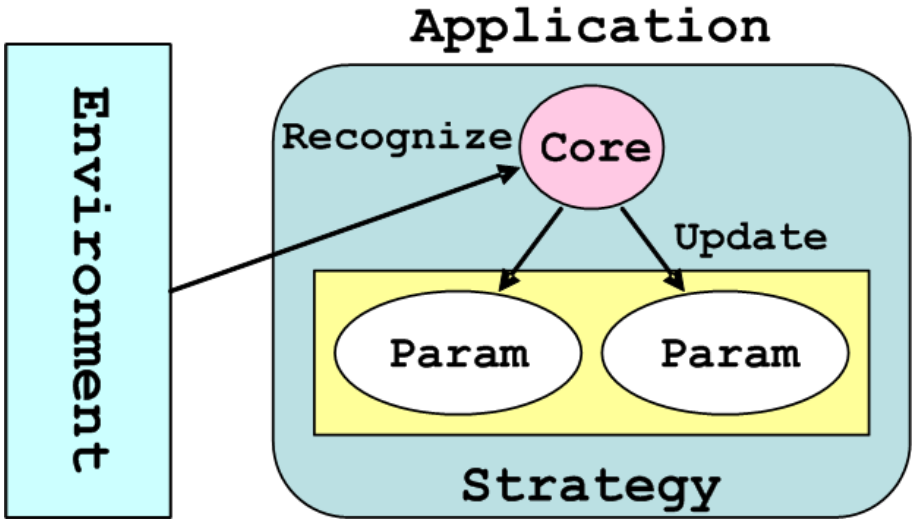

Figure 1.2: Parameter adaptation

On the other hand, the compositional adaptation mechanism can be used to directly replace the algorithmic or structural system components. TCP can change its behavior as the environmental changes, but there are some cases where TCP is not suitable. For example, Tian et al.[6] pointed out that the traditional TCP performs poorly in a wireless environment due to its inability to distinguish the packet losses caused by network congestion from those attributed to transmission errors. Therefore, such as TCP-Westwood[7] and TCP-Jersey[8], a revised TCP that has mechanisms for deciding the cause of a packet loss and for accurately estimating the network bandwidth is needed for creating a more efficient wireless communication.

Next, let us take the encryption algorithm for another example; encryption is the basis for current networks and there are many standardized algorithms. However, as there is an ever-increasing amount of computational power and algorithm analysis, most algorithms cannot gradually ensure their amount of security. Therefore, more powerful algorithms need to be created to replace the old ones in many network systems, such as AES[10] that has replaced DES[9]. Consequently, networking softwares has to use a variety of functional strategies to suit the individual network environments, and a compositional

adaptation mechanism is essential for this purpose.

Figure 1.3 illustrates the conceptual diagram of a compositional adaptation. Compared with parameter adaptation, additional programs are needed because most applications do not have a mechanism that can dynamically and automatically replaces some program components. A *common platform* is a kind of execution environment for applications. Functional strategies are realized as components that are stored within the platform, and these components can be replaced with an application's default components at runtime. To support a variety of applications implemented with different API and ABI, a common platform must provide abstract interfaces to both the users and the applications. An *adaptation mechanism* also checks the environment and determines whether the current component is an adequate strategy. If not, the adaptation mechanism replaces the default strategy with another one offered by a driver program. Generally, since the application and adaptive mechanism are structurally divided, a special interface that is like glue interface is required to attach the adaptation mechanism to the application.

1.3 A Transparent Approach to Compose Adaptive Software

Adaptive software is an effective method to keep up with the ever-growing network environment, but it is impossible for every piece of networking softwares to be made adaptive. As I mentioned before, backward compatibility must be considered in order to widely deploy new mechanisms in the network. Therefore, a transparent approach for making existing networking software adaptable must be developed in order to keep every piece of software up-to-date. In particular, the transparency would mean that the application's capabilities would be extended without the developers having to statically modify the source code. Although this transparency seems difficult to achieve, several software techniques have been proposed for the transparent extension of the networking software capabilities.

Table 1.1 lists some transparent extension techniques. Although there are many tech-

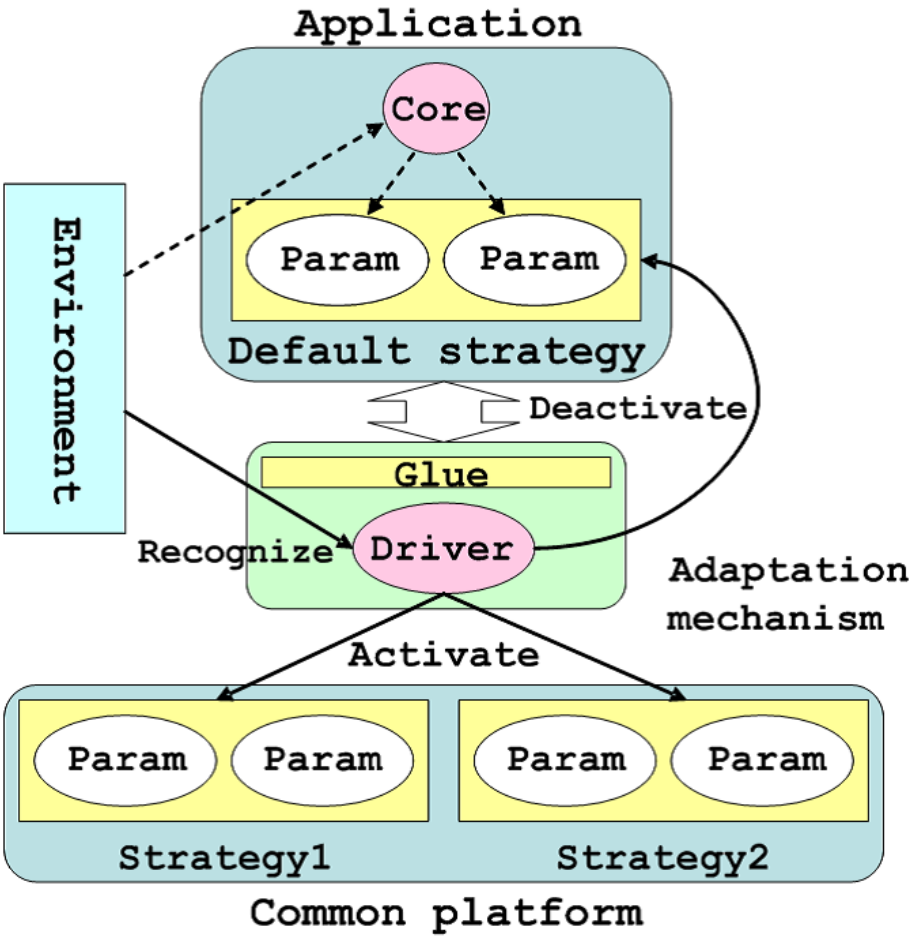

Figure 1.3: Compositional adaptation

niques, an adequate one has to be used to compose adaptive networking software so that the effort for introducing the adaptive mechanism will be minimal, and as many systems as possible can be supported.

1.4 End-to-End Consistency of Adaptive Communication

Many researchers have thus far proposed various adaptive software and transparent extension techniques. However, little attention has been given to maintaining end-to-end consistency during adaptive communications. In particular, the interoperability with existing networking systems has not been discussed. Since adaptation implies that the composition of a protocol stack is changed, the other protocol stack must be similarly

Table 1.1: Transparent extension techniques of networking software capabilities

Approach	Description
Language-based	Extension programs are implemented within the source code or runtime component automatically. Static extension method like Aspect Oriented Programming and dynamic extension method like dynamic binding are commonly used.
Binary-based	An additional system like middleware alters program's behavior by modifying API's internal behavior using interposing techniques, or directly changes application executables using process image manipulation mechanism.
OS-based	The protocol stack is implemented within the kernel in many OSes. Therefore, every networking functions can be manipulated as long as the OS allows modification of the protocol stack.
VM-based	Virtual machine provides real network environment to the guest OS and applications by bridging real and virtual network. Since VM can gather network packets from/to applications, it is possible to introspect and modify them by software.
Proxy-based	Proxy is an independent program located between end-to-end systems. Proxy has a capability of analyzing protocol layers, and can act as a protocol interpreter. There are host type proxy and network type proxy.

changed at the same time. Therefore, existing studies have implied a local adaptation where the implementation itself is updated at one end while maintaining the same protocol interface, or that develops its own control protocol that conveys messages for adaptation.

Local adaptation can be introduced into existing networks because the remote appli-

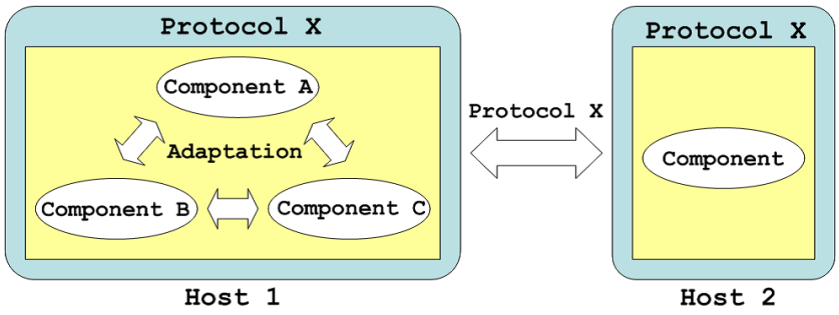

Figure 1.4: Local adaptation

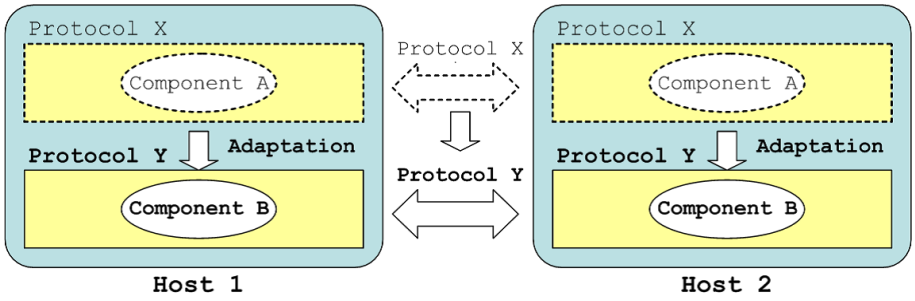

Figure 1.5: Global adaptation

cation does not need to follow the adaptation (See Fig.1.4). On the other hand, global adaptation requires a mechanism to maintain an end-to-end consistency during the adaptation (See Fig.1.5). Considering the interoperability with existing networks, the consistency mechanism should be realized on common protocols or higher-level protocols such that existing network equipment can deal with the communication flow.

1.5 Purpose and Composition of This Paper

I would like to explore a common software framework in this paper for diversifying the networking capabilities by focusing on the architectures of networking software, such as the applications, servers, and protocol stacks. The proposed framework intends on being used in networking applications or servers to extend their networking capabilities. This study takes into account the methodologies and implementation techniques for extensible and interoperable networking softwares. In particular, instead of creating a brand-new network environment, existing network environments are enhanced such that advanced

services are widely introduced by making the network software functions more flexible within the compatibility constraints.

Network applications usually use socket interfaces that abstract networking functions to communicate over the network. A socket interface generally supports two types of communication styles: connection-oriented communications and connectionless-oriented communications, and it enables for the construction of network software to be independent from each network protocol. So, I am proposing a system that transparently brings the composability and adaptability of the networking capabilities to applications by focusing on the abstraction of the socket interface.

In particular, a generic framework called FreeNA is proposed as an instantiation of a compositional adaptive system. FreeNA hides the platform-dependent issues like API and ABI (Application Binary Interface), and also offers an abstract interface to its users and applications. The networking capabilities of the application are adaptively composed by transparently alternating the behaviors of the socket functions. Therefore, users can compose the application's capabilities in order to meet the networking environment without having to directly modify the application's code. In addition, since FreeNA provides a higher-level abstraction such that the users can instruct the networking capabilities and environmental conditions using a human-readable configuration file, FreeNA can be used to construct autonomic computing environments.

So far, many related systems have been developed that can transparently change the networking capabilities of an application. However, the existing systems have some restrictions on the platform, performance, developer-oriented behavior, usability, flexible functions composition, backward-compatibility with existing network environment, and end-to-end consistency. Compared with the existing studies, this study aims at a more pragmatic system by overcoming the drawbacks of a realistic environment such as the operating systems, programming languages, and network protocols.

The rest of this paper is organized as follows.

Chapter 2: Related Work for Adaptive Systems and Communications

We start by introducing the current studies for adaptive systems, their transparent approaches, and networking services. First, adaptive systems are categorized by their transparent approaches. Their characteristics and drawbacks are also described from the perspective of their flexibility, interoperability, and usability. Next, the practical software services and support systems are presented as the basis of the adaptive communications. Finally, the related systems are reviewed from the perspective of the protocol composition and end-to-end consistency compared with the proposed system.

Chapter 3: Fundamental Policies for Adaptive Communications and System Usability

The fundamental policies for the proposed system are considered to represent a vision for the above-mentioned goals. The purposes of FreeNA are clarified first. Based on these purposes, the expected users and usage scenarios are described. Next, the characteristics that FreeNA needs are considered. Moreover, some network services examples are also given to show the usefulness of FreeNA.

Chapter 4: Architectural Design and User Experience

The architectural design of FreeNA is explained in light of the FreeNA characteristics described in chapter 3. Since FreeNA is composed of several subsystems, the role and relationship to the other components are clarified for each subsystem. In addition, the actual usage of FreeNA with the configuration file for the service composition and operational commands is also described to prove the usability of FreeNA.

Chapter 5: Implementation

The internal architecture of the FreeNA system is clarified. First, the implementations of the FreeNA client and server are explained, then the mechanism for the service insertion and the implementation of the flow handler are explained. In addition, the mechanism for the Transport-layer protocol insertion is also described in addition to that of the

upper-layer service insertion.

Chapter 6: Evaluation

The FreeNA system is evaluated on the basis of several diverse perspectives to prove the fulfillment of FreeNA's goals. The functionality of FreeNA is compared with that of other similar systems first. Next, the manageability and security aspects are also discussed. Furthermore, the system performance of FreeNA is measured. In particular, the performance overhead of the service inserting mechanism itself is evaluated, rather than the effects of the network service execution.

1.6 Summary

The characteristics and issues of the current network environment as background of this paper are described in this chapter. Current networks that are based on TCP/IP have became social infrastructures that include the Internet, and still evolve with the introduction of advanced services. However, TCP/IP only provides the fundamental connectivity on the network, but more practical systems must support advanced and diversified networking capabilities on their own.

Some key technologies for keeping up with this evolution are autonomous computing and compositional adaptation. Autonomous computing is a concept where users can easily manage computer systems using higher-level human guidance. Compositional adaptation makes software adaptable such that the software dynamically modifies its structure and behavior in response to changes in its execution environment. If widely-used networking software becomes compositionally adaptable using higher-level human guidance, a variety of advanced network services can be incorporated into large-scale networks without incurring the backward-compatibility problem. In addition, we must consider the end-to-end consistency to leverage these methodologies for adaptive communications.

In this paper, I propose a common software framework for diversifying the networking capabilities. The proposed framework FreeNA intends the networking applications or servers to extend their networking capabilities by using the compositional adaptation

method. FreeNA hides the platform-dependent issues like API and ABI, and also offers an abstract interface to its users and applications. The networking capabilities of the application are adaptively composed by transparently alternating the behaviors of the socket functions. Throughout this paper I take into consideration the more essential points for achieving a more practical adaptive communication mechanism by discussing FreeNA.

Chapter 2

Related Work for Adaptive Systems and Communications

The proposed framework, named FreeNA, is a common platform for a variety of networking applications and servers for adaptive communications and autonomous computing. Since these two methodologies are based on various technological elements and conventional aspects, many existing studies are related to this research.

In this chapter, the existing studies on adaptive systems, their transparent approaches, and their networking services are introduced. First, the adaptive systems are categorized by their transparent approaches. Their characteristics and drawbacks are also described from the perspective of the flexibility, interoperability, and usability. Next, practical software services that can be incorporated as higher-level network protocols and support systems are presented as the basis for adaptive communications.

2.1 Compositional Adaptive Systems and Their Methodologies

The automatic transformation of an application's capabilities is the heart of compositional adaptation. Ideally, every application has an adaptive mechanism to fit the execution environment because it takes a lot of effort for developers or administrators to keep up with all the necessary changes. However, considerable knowledge and development skills are required to develop adaptive systems. For this reason, an additional mechanism is needed to incorporate an adaptive mechanism into existing systems. So

far, many researchers and engineers have developed support systems and techniques for the necessary adaptation. As a result, several types of recomposition techniques for the software capabilities have been recognized as the fundamentals of software adaptation.

I describe the existing adaptive systems and recomposition techniques in this paper, and their characteristics and drawbacks are also discussed.

2.1.1 Language-based Adaptation

Extension programs are automatically implemented within a source code or runtime component. Generally, there are two types of adaptation, compile-time adaptation and runtime adaptation. In compile-time adaptation, extended functions are directly inserted into the source code of the target application using Aspect-oriented Programming (AOP) method[11] or Subject-oriented Programming. However, runtime adaptation systems insert extended functions using dynamic binding mechanisms provided by the language itself. In particular, a dynamic object loading mechanism, an introspection method, a byte-code manipulation technique, and a VM modification technique are used.

Source Code adaptation

AOP was thought up based on the *Separation of Concern (SoC)* in order to weave *cross-cutting concerns* into the specified point automatically at the compile-time. To construct highly reusable software components, environment-dependent codes (aspects), such as those for security or logging, have to be separated from the core component. Thus, AOP enables developers to implement aspect codes as independent components and to specify the join-point into the core component. As a result, an aspect compiler automatically weaves the aspects into the join-point before compilation.

Zhang et al.[12] proposed a model-driven approach to introduce an adaptive mechanism to non-adaptive legacy systems while maintaining the use of the AOP mechanism. In this approach, three conceptual levels are introduced to provide an adaptation with assurance. First, the fundamental adaptation is applied using the UML model of the existing system. Next, the correctness of the adaptation is checked using the formal method from

the perspective of the local properties and global invariants. Finally, an extended code is implemented and added using the AOP technique. As a result, non-adaptive existing systems can radically and correctly become adaptable systems.

MetaSockets[13][14] are adaptable communication components that provide a reflective extension to the existing Java classes. MetaSockets are created from the existing Java socket classes by absorbing their functionalities and interfaces using Adaptive Java[15]. That is, MetaSockets provides a higher-level abstraction of the networking capabilities, in contrast to the socket interface only providing a lower-level abstraction similar to the network connectivity. Since MetaSockets provides the same interface as the absorbed socket classes, the existing Java classes can invoke traditional socket operations using MetaSockets' functions. Adaptive Java enables runtime adaptation of the internal architecture of MetaSockets and their behaviors in response to the environmental changes at runtime. Therefore, users can make their Java applications adaptable without modification of their core components.

Runtime Environment adaptation

ASM[16] is a Java class code generation and manipulation tool designed to dynamically generate and manipulate Java classes. Although ASM is based on a *visitor design pattern*[17], it does not de-serialize the object graph, and directly modifies the serialized graph directly. In this approach, there are there three objects for instrumentation; one class analyzer and two visitors, and they collaborate with one another to dynamically manipulate the instruction sequence. As a result, the runtime applications can be small and fast because unnecessary de-serialization is not executed per bytecode instruction.

Javassist[18] is a Java bytecode manipulation tool based on the reflection mechanism. While other similar systems provide a behavioral reflection that interposes the method invocation to alter its behavior, Javassist supports the structural reflection that can modify the definitions of the data structures, such as the class, method, and record. Javassist manipulates the bytecode of the Java class when the class is loaded to Java VM. In particular, the advantage of this method is that Java VM does not need to be modified.

In addition, users can modify the behavior of the specified Java method by replacing the original method with another one. Therefore, it is possible to introduce an adaptive mechanism into existing Java programs without modifying them.

Problems and Perspectives

Although these systems enable a granular program enhancement mechanism, users of these systems are largely limited to developers because the users have to know the internal architecture of the system and its behavior in detail. Even though there are experienced developers, it is not easy to analyze a large-scale system and develop an appropriate adaptation strategy, and normal users obviously cannot involve themselves in these methods.

In addition, users must use the support system corresponding to the programming language that is used to implement the target application. Therefore, there are cases where users cannot find adequate systems for their target applications.

As for the aspect-oriented programming and subject-oriented programming techniques, there is no unified aspect/subject-oriented system, and the functionality and usability are different from each other. Therefore, users can only use aspect/subject-oriented system corresponds to the target language, and it is difficult to widely apply these methodologies to a variety of existing systems. In addition, users must have the source code of the application. However, there are common cases where the source codes of the applications are hidden like many commercial systems.

On the other hand, runtime environment adaptation systems like ASM and Javassist do not require the source code of the target application. However, these systems target more experienced developers who have internal knowledge about the language systems in order to manipulate the intermediate language.

Compared with these systems, the proposed FreeNA system also targets inexperienced users by offering a highly abstracted interface. That is, there is no need to write a code to incorporate other components into existing systems. Moreover, the source code of the target application is not needed to change the existing application's behavior because

FreeNA uses a binary-based adaptation method that will be described later in this paper.

2.1.2 Binary-based Adaptation

In binary-based adaptation, the application's behavior is altered after compilation without changing the source code. Two types of manipulation techniques, system-call interposition and process image manipulation, are commonly used for the modification process. In both techniques, since the source code cannot be modified, another program like a middleware system is needed to handle the target application.

System-call Interposition Techniques

Basically, applications have to call system-calls or API to use the OS's capabilities, such as the networking and file I/O. That is, any type of application implemented with different programming languages eventually uses the same system-call as long as the underlying platform is in the same series. Therefore, it is possible to interpose the system-call invocation and alter its internal behavior by preparing a hook at certain system-call execution points.

An Interposition Agent[19][20] is a higher-level or abstracted toolkit for effectively using the system-call interposition. This toolkit interposes an arbitrary user code expressed as high-level objects between the target application and the instances of the system interface, rather than the directly modifying the intercepted system-call. Users can implement additional codes as objects that have several functions corresponding to the traditional system-calls. System-call interposition is achievable by using a special system-call, `task_set_emulation()` that is offered by the Mach operating system[21], which redirects the instruction point to a specified user code when the specified system-call is invoked. Therefore, the users' applications can be used as they are as long as they use the corresponding system-calls.

DITOOLS[22] is an application-level tool that supports the dynamic interposition on dynamically-linked function call boundaries. DITOOLS appends additional execution stages, which are the extension and binding stages after a loader's reference resolution. In

the extension stage, the need for extending is checked based on the user configuration file, then the bindings between the dynamically-linked references and definitions exported by the modules are changed in the binding stage. To arrange the execution stages and manipulate the executable images, a customized runtime linker and loaders are introduced. Therefore, these tools can be applied to not only the system-calls offered by the platform, but also the library functions that are dynamically linked. In addition, users can use DITOOLS without privileged user intervention because this tool works in the user-space.

A library preloading technique can be used to replace the entire dynamic-linked library with a specified one at the time of execution. Many platforms provide some mechanisms that the runtime linkers can check for specified preloading libraries before linking the default libraries. For example, the Unix/Linux platforms offer `LD_PRELOAD` or a similar environmental variable, and the linker checks this variable to determine the involved libraries. Therefore, this technique can only be applied to dynamic-link libraries, so static-link libraries or functions cannot be changed. System-calls are invoked by triggering an architecture dependent software interruption mechanism instructed with an assembler language. However, since many system-calls are wrapped in a user-level system library, the library preloading method can be used to modify the internal behaviors of system-calls. In practice, TESLA[23], Trickle[24], and TCP Stream Filtering[25] use this technique to interpose socket functions to execute additional user codes.

FUSE[26] is a framework that supports implementation of a new filesystem in the user-space for the Unix/Linux platforms. FUSE consists of an API and a loadable kernel module. Users can define their file I/O functions that have the same interface with traditional system-calls within a `fuse_ operations` structure, and it is used by the API offered by FUSE. FUSE's kernel module is loaded into the kernel-space to hook into the Virtual File System (VFS) and redirects the file I/O system-calls to the user-defined functions. User filesystems are implemented as executable binaries linked with the FUSE library. Consequently, the applications and underlying kernel can use full-scale additional filesystems without any modification.

Process Image Manipulation Techniques

Instead of system-call interposition techniques hooks between the applications and kernels to intercept a function call, the following systems directly changes the code instructions or binary executables. Since process image manipulation also allows users to introspect and modify the internal structure of the program, such as the text code and memory variables, the application's behavior can be altered more radically. In addition, not only the dynamic-link functions but also the statically-linked functions can be intercepted. Therefore, it is possible to develop an application-dedicated intermediate system that finely manipulates the application without the source code.

Vx32[27] is a multipurpose user-level sandbox system for the x86 architecture that enables any application to safely load and execute extended code. Vx32 traps the I/O instructions of the host application to prevent malicious codes from being executed, and it confines guest codes to a particular memory region in order to separate them to the host application's memory region. Since Vx32 works entirely within the user-space, it cannot use a special kernel mechanism or the processor's privilege-level mechanism. Therefore, it leverages the dynamic instruction translation techniques like Valgrind[28], which directly and safely rewrites the code sequence before executing it. Since many operating systems allow users to access the x86 architecture capabilities like the segmentation mechanism, vx32 can work on many common kernels, such as Linux, FreeBSD, and Mac OS X without needing special privileges and kernel extensions. Moreover, extended codes can be executed efficiently because the application's instruction sequence is directly modified.

Dyninst[29][30] is a C++ class library and API for post-compiler program manipulation. DyninstAPI can change an application's process image already running on many platforms, such as Linux, FreeBSD, Solaris, AIX and Windows by dynamically instrumenting the code into the image. One characteristic is that the C++ classes and the API are represented as a machine-independent mechanism even though a binary manipulation technique heavily depends on the underlying platforms and machine architectures. Therefore, users can use the same classes and API irrelevant of their platforms. Similar

to Vx32, Dyninst can directly arrange a code sequence so that it introspects/modifies all the symbols within the image, even for variants, functions, and even dynamically linked libraries. Moreover, additional code blocks can be inserted into a specified instruction point, and particular instructions can be removed in their entirety from the image. As a result, the users can completely control the application's behavior and internal structures with Dyninst.

Livepatch[31] is a user-level intermediate tool for applying binary patches to already running applications on Linux. Livepatch enables users to manipulate the application process image for setting arbitrary values to specified memory addresses, allocating extended memory space within the process image, and linking and loading extra shared libraries. Actually, Livepatch utilizes the `ptrace()` system-call, which can intercept specified system-calls before and after execution, and the BFD library[32] that can handle a variety of binary formats. Since `ptrace()` has to hook a specified system-call per invocation each time, it causes significant overhead when the system-call is executed many times.

Problems and Perspectives

Although these systems and tools can be used without the source code of the application, recompiling, and relinking, their mechanisms are mostly platform-dependent, and it is impossible to support so many platforms and machine architectures by using a single tool even though there are some portable systems like Dyninst. Therefore, several binary-adaptation techniques should be combined to widely introduce an adaptation mechanism into existing systems.

Special system-calls for an interposition, such as `ptrace()` and `task_set_emulation()` are reasonable enough to modify an application's behavior at runtime. However, the performance overhead of the invoking user codes is relatively large because the execution flow is moved from the application to the kernel to redirect the specified system-call. Consequently, a lot of context switching between the user-space and kernel-space are needed to execute an extended user-code.

Customized runtime linkers and loaders are effective at rebind function calling during runtime. However, this binding can only be applied to dynamically-linked functions not statically-linked functions. In addition, it is difficult to port these systems to other environments because the runtime linker and loader are core systems of the runtime environment and so many existing programs depend on them.

The library preloading technique is also a simple method for interposing additional codes into the execution flow of the library function. However, this technique can only be applied to shared libraries too. In addition, it is difficult to compose a cascade of several libraries independent of each other. To do this, a mechanism that can create a glue library with dependency information is necessary. Moreover, users have to know the library information, such as the exporting functions and versions.

On the other hand, the process image manipulation technique can directly change the behavior of the application, and not only the library calls but also the statically-linked functions can be handled without needing special platform capabilities. In addition, this technique enables for the efficient execution of the user codes because additional instructions are not needed for the interposition. Generally, the structure of a process image varies because of the object format and processor architectures, and advanced techniques are required for users even though the abstract interface is provided by Vx32 or Dyninst.

In contrast, FreeNA is designed for portable systems even though it uses binary-adaptation mechanisms. This portability is achieved by separating the platform-dependent techniques from the core systems of FreeNA, and by implementing a mechanism that can select an adequate adaptation mechanism depending on the user's platform. In addition, FreeNA provides more higher level abstract interfaces to its users, rather than hiding the platform-dependent issues. Therefore, users can concentrate on interposing user codes without having to take into consideration the interposition method.

2.1.3 OS-based Adaptation

Generally, common operating systems directly manage the resources such as the network access, and hides a low-level mechanism from the users and applications. OS-level adaptation can control not only an application's behavior but also the network capabilities in more detail than other adaptation mechanisms. However, many OSes are not designed to extend their internal structures and modify their behaviors using users' operations. Therefore, dedicated systems are needed to achieve the user-oriented recomposition of the OS capabilities.

x-Kernel[33][34][35] is a composable OS for network protocols where users can configure and compose the modularized protocols within a kernel during the operation time. Implementations of network protocols are abstracted as object-oriented designs, and each protocol object corresponds to a conventional network protocol where the relationships between protocols are flexibly defined. The relationships between protocols are defined as a graph structure with user-oriented tools, and `C` code is automatically generated from the graph. That is, users can implement their desired protocols as independent modules, and configure their composition and relationships using the dedicated tools within the kernel. Currently, *x*-Kernel is ported to other platforms as a user-level virtual operating system in order to deploy flexible networking capabilities mechanisms into a practical environment.

Stream[36] is a component-based I/O mechanism on the Unix system V that linearly connects applications and devices like the pipe mechanism. The stream is composed of several independent modules that express the communication capability, and each module receives messages from the previous module, processes them within the module, and passes them to the next module in both directions. Although the two end modules in a stream are automatically connected to work as an interface with the application and the device, intermediate modules can be dynamically attached upon request by the application. Therefore, network protocols can be implemented as stream modules, and the users can directly introduce new protocols into the protocol stack within the kernel. Like the socket interface, an interface to a stream is provided as API – TLI/XLI and the

necessary networking capabilities are decided by the application through this API.

Scout[37][38] is a subsequent *x*-Kernel OS that can also adapt the changes of the execution status of the kernel. Scout introduced a **path** concept that expresses —. The notion of the path enables the dynamic adaptation of a system resource allocation or scheduling based on the application's characteristics, such as the IO-intensive.

Problems and Perspectives

Although the operating system supports an effective and efficient way for composing networking capabilities corresponding to its environment, it is difficult to introduce this mechanism to widely-spread existing systems. First, the common operating systems currently in widespread use, such as Windows and Linux, do not have user-level flexible composition mechanisms for their kernel capabilities. As a result, dedicated composable mechanisms have to be ported to practical OSes as a kind of middleware in the user-space like the current *x*-Kernel, and the host OS's functions should be diverted to use the low-level mechanisms. Consequently, it becomes difficult to avoid competition between the host OS's capabilities and the user-space OS's. Moreover, since applications have to use the specialized API provided by these OSes to configure the compositions of the capabilities, the already existing applications cannot use these mechanisms without modification. In addition, the composition of the capabilities within the kernel requires a rebooting of the system, so the application cannot immediately adapt its executing environment.

2.1.4 VM-based Adaptation

A virtual machine provides an abstract and practical machine architecture for the guest OS. The guest OS's and applications' behaviors are dynamically altered such that particular code instructions are translated to fit with the authentic machine architecture. Generally, three modification techniques, binary rewriting, para-virtualization, and hardware support, are used to execute the code instructions for the host machine without modification of the guest OS and applications. Therefore, it is possible to add a hook into the virtualization mechanisms in order to execute specified user codes. In addition, a

virtual machine provides a real network environment to the guest OS and the applications by bridging the real and virtual networks. Since VM can gather network packets from/to applications, it is possible to introspect and modify them by using software.

VTL[39] is a framework for composing a virtual traffic layer onto existing virtual machines. The virtual traffic layer enables for packet modification and creation to transparently modify the network traffic to the VM and its applications. VTL consists of several types of API, such as packet acquisition, inspection, modification, and serialization. These APIs are used to implement networking services as independent modules. VTL uses the libpcap[40] or Winpcap[41] packet capture the libraries to capture packets transferring from/to the VM and the libnet[42] library to inject additional packets into arbitrary packet streams. Therefore, VTL can be applied to VMs that provide a virtual networking interface accessible to pcap and libnet without having to modify of the VMM, host OS, and applications. Moreover, users can modify the lower-layer networking capabilities like the TCP protocol because libpcap can capture packets containing whole headers.

PinOS[43] is an extension system of the Pin[44] dynamic instrumentation framework. Pin provides user-level code instrumentation APIs to transparently modify the application's behavior. Compared with similar software instrumentation frameworks, PinOS has two major advantages – it can instrument not only user-level codes but also kernel-level codes, and it is designed to support multiple platforms including Linux and Windows even though it can currently support only Linux. In order to achieve both two characteristics, PinOS is based on software dynamic translation technology using a modified version of Xen[45] VMM and Intel VT technology[46]. With the virtualization mechanism, PinOS can use the system facilities of the host OS without any potential re-entrance problems because the kernel's instrumented codes and the instrumentation engine are completely completely separate. In practice, PinOS is inserted between the guest OS and Xen VMM so that PinOS will be in a position to observe every instruction executed in the subject system, and a dedicated Xen driver for PinOS is installed into Xen to provide several

system functionalities, such as memory stealing, attach/detach, I/O services, and time virtualization. Therefore, all the system instructions can be fully controlled by PinOS without any modification.

Problems and Perspectives

Virtual machines (and hardware virtualization mechanism) enable system-wide virtualization. In other words, even the low-level facilities of the kernel and resources can be controlled as long as VM supports the user's customization. As described, VM-based systems interpose user codes where the execution flow or data flow is concentrated.

Packet capturing on VM is a simple method to entirely acquire the intended data because many VMM have the interface to the physical network and capturing libraries can attach to it. However, this method has two drawbacks. One is that the analysis and reconstruction of captured packets are required to modify their protocols. The other is that packet manipulation will be executed at the user-level unless a user-defined capturing and manipulation mechanism is implemented within the VMM. Therefore, the performance overhead of packet capturing and manipulation is significant, as shown in VTL system.

Binary instrumentation on VM is one of the few methods that transparently modifies both the user programs and kernel facilities. Since VMM hides the physical machine architecture and provides an abstract architecture to the guest OS, a variety of OSes can be operated on the VMM. That is, the entire system behavior can be altered if VMM contains arbitrary interposition mechanisms in the user's configuration. However, the internal architectures and abstraction mechanisms of VMM vary, and some systems require hardware virtualization support. Therefore, this method can only be applied to a confined VMM and physical machine architectures.

2.1.5 Proxy-based Adaptation

Proxy is an independent program located between end-to-end systems, and works as an intermediate client or server in order to split end-to-end connections or modify the content of the communication. Therefore, proxy has the capability of analyzing relevant

protocol layers, and can act as a protocol interpreter. By using the proxy mechanism, it is possible to completely modify the networking capabilities of the client or server systems without modification of the application program and OS.

Stunnel[47] is an independent proxy software that tunnels the TCP connections into the SSL/TLS connections. Stunnel can work on the client-side or server-side proxies in order to communicate with an SSL-compatible application instead of a non-SSL compatible application. Therefore, two connections are used to link both applications, a connection between non-SSL compatible applications and Stunnel, and the other connection between Stunnel and SSL-compatible remote applications. To achieve this connectivity, Stunnel has to deconstruct the original messages from a packet stream, and reconstruct the transformed messages into the packet stream. Since Stunnel does not intervene in the internal structure of the application and the kernel, it ensures the that transparent adaptation of their behaviors.

A^3 (Application-Aware Acceleration)[48] is a middleware that offsets the behavioral problems of applications, such as thin session control messages or block-based data fetches. A^3 aims to boost the performance of wireless communications by controlling the application-layer protocols session so that redundant and aggressive retransmissions, prioritized fetching, infinite buffering, and application-aware encoding are transparently completed, rather than using the wireless-version TCP protocol. Internally, A^3 consists of a user-space software module for the mobile clients and a proxy network appliance for the servers. On the client side, the module captures the incoming/outgoing packets using the NetFilter utility in Linux[49] or Packet Filtering interface in Windows[50], and the captured packets are manipulated at the IP-level. Although every packet can be flexibly captured and manipulated at lower-level, packet capturing methods require a lot of execution time and result in a lower level of performance in actual environments.

Prometeo[51][52] is a multi-function and extensible modular-proxy system based on the Internet Daemon[53] concept. Prometeo enables users to incorporate various networking services, such as the IPv4/v6 transparent support, SSL tunneling, logging, and transpar-

ent proxy support, into Prometeo simultaneously in a manageable way. These services are implemented as independent modules using a dedicated C++ framework. Therefore, users can flexibly compose their intended networking capabilities without building many different proxy systems. Actually, Prometeo adds an abstraction layer between the application logic and the traditional socket interface within the whole application program by encapsulating all the required functions in a set of C++ classes, rather than working as an independent network node or middleware system.

Problems and Perspectives

Rather than changing the software structures and behaviors, the proxy-based approach attaches data flows to transform the data itself directly. There can be two types of proxy systems, a fully-independent proxy like Stunnel and a semi-independent proxy like Prometeo.

A fully-independent proxy can transparently interpose an adaptive mechanism between the end-to-end systems, and each end system is unaware of the existence of the proxy. However, there are some cases where this type of proxy cannot be dealt with. One is when an intermediate proxy splits the end-to-end connection, and does not ensure the end-to-end connectivity and transparency[54]. To link both end systems, the proxy system must guarantee their connectivity. For example, the communication port is determined dynamically like in FTP-data, and the proxy cannot know the port number unless it analyzes the application protocol directly, and proxy systems do not reflect the special functionalities of the lower-layer protocols like the flow control even if the end system uses such functions. In addition, network throughput will be significantly decreased because the application data are extracted from the packet stream once and recomposed as another packet stream after the manipulation.

On the other hand, the semi-independent proxy system is weakly attached to the end system even though the proxy is a different executable program from that of the target application. For example, end-to-end connectivity and transparency can be ensured by the proxy controls of the the end application's network processing. As a result, the proxy

properly knows the intended behaviors of the application. However, even though the extended networking capabilities are not recognized from the end application, the proxy has to let the application use the proxy's functionality.

2.2 Practical Networking Services and Systems

By using the transparent adaptation mechanisms described thus far, a variety of networking services types can be incorporated into the existing system flexibly. Regarding the networking services, there are many types of functionalities and working protocol layers. Here, representative networking services for adaptive communications and their support systems are described in separate protocol layers.

2.2.1 Upper-layer Networking Services

TESLA[23] is a common platform for session-layer services. One service sets up multiple TCP connections between end-to-end applications to improve the throughput of a single logical data transfer. Another service migrates the end-to-end sessions for mobility when the original session is disconnected. Another service shares congestion information across multiple flows sharing the same network path in order to control the network congestion by using compression, application-layer routing, or traffic shaping.

Trickle[24] provides the portable solution of an ad-hoc rate limiting mechanism in the user-space. The Trickle library is inserted between the application and the **libc** library where the network socket functions are implemented within, by using library preloading technique. The rate limitation is performed by either using a delay of the I/O process, truncating the length of the I/O, or a combination of the two according to a scheduler that controls the transfer rate. In addition, Trickle allows for a collaborative rate limitation across multiple flows using the global rate limitation and priority rate limitation based on the communication type.

Agetsuma et al.[55] developed a mechanism that transparently converts one application-layer protocol to another using mobile codes. When a server supports a new protocol, for example SMTP is upgraded to SMTP-AUTH, then mobile codes used as a protocol

converter are downloaded and automatically converted the old client protocol. Then, the client can transfer the mail to the server using the SMTP-AUTH protocol. Generally, mobile code exploiting might incur security problems because malicious programs pretending to be useful could be unawarely executed. Therefore, secure mechanisms are also introduced to protect the client system. First, the mobile codes to be downloaded are authenticated using a digital signature or trusted site. At runtime, the mobile codes are executed within the secure sandbox that controls the resource accesses from the mobile codes.

2.2.2 Lower-layer Networking Services

Alpine[56] allows for the development of transport-layer protocols within the user-space instead of the traditional protocols within the kernel. To avoid kernel modification, Alpine moves all the network stack functions including the socket interface from the kernel into the user-space libraries. That is, the code of the protocol stack is recompiled as the user libraries in order to ease the process that the newly developed protocol has incorporated into the kernel protocol stack. In addition, a software-only network interface driver is introduced to the transmit/receive raw packets bypassing the kernel protocol stack. The drawbacks of Alpine are that it can be a highly platform-dependent system even though it does not modify the kernel codes.In this system, the transport-layer protocol has to be properly developed to keep within the kernel specifications, such as the kernel data structures and the synchronization mechanisms. Next, the actual throughput of Alpine will be highly decreased because the whole networking process is executed within the user-space, and due to its implementation style.

DR-TCP[57] supports the on-the-fly reconfiguration of the TCP protocol at runtime by downloading another TCP implementation without compiling the code of the protocol, and restarting the application and the system. The DR-TCP protocol is designed based on the recursive state machine model and object delegation in order to model the protocol functions separable from the states, and delegate the state machine's event handling to another state machine. That is, one state machine is seamlessly translated into another

state machine that represents different TCP configurations. In that paper, the DR-TCP protocol was converted to TCP-Westwood by translating the establishment, retransmission, and fast recovery functions. DR-TCP is implemented in the user-space using a raw socket mechanism. However, a kernel function that passes the received IP packet to the TCP processing function is modified to replace the kernel TCP protocol with the DR-TCP protocol.

2.3 Summary

In this chapter, the existing studies related to adaptive communication systems and their mechanisms that introduce additional functions were explained. Moreover, the drawbacks of these systems were pointed out in view of their functionality, implementation, performance, and practical usage.

So far, many researchers and engineers have developed support systems and techniques for adaptation techniques. As a result, several types of recomposition techniques of the software capabilities have been recognized as software adaptation fundamentals.

Language-based adaptation techniques including aspect-oriented programming enable a granular enhancement of the target applications. However, users of these systems are largely limited to developers because they have to know the internal architecture of the system and its behavior in clear detail.

Although binary-based adaptation techniques including system-call interposition do not require reconstruction of the applications themselves, their adaptation mechanisms are mostly platform-dependent. The library preloading technique is widely used to interpose additional codes into the execution flow of the library function on many platforms. However, this technique can only be applied to shared libraries, and it is difficult to compose a cascade of several libraries independent of each other.

OS-based adaptation techniques support the effective and efficient way to compose networking functions. However, this approach restricts the available systems and complicates the management of the system.

VM-based adaptation techniques can introduce a variety of extended functions including low-level to high-level functions by using packet capturing or binary instrumentation. However, the packet capturing method requires difficult adaptation tasks and performance overhead. Moreover, binary instrumentation can only be applied to a confined VMM and physical machine architectures.

Proxy-based adaptation techniques can be used to attach data flows. A fully-independent proxy can transparently interpose additional codes between the end-to-end systems. However, end-to-end connectivity and transparency are not ensured, and the network throughput will decrease. A semi-independent proxy system is weakly attached to the end system. However, the proxy has to let the application use the proxy's functionality.

Furthermore, practical networking services and systems that utilize introduced techniques to insert services were also described. These services were included in the upper-layer networking services, such as for ad-hoc flow controlling and protocol upgrading, and lower-layer services, such as another transport-layer protocol in the user-space. These services and system can be useful for creating a FreeNA system.

Chapter 3

Fundamental Policies for Proposed Systems

As described in chapter 2, many methodologies for adaptive systems and networking services have already been proposed. However, there are still many problems that need to be overcome in order to apply an adaptive mechanism to current widely deployed network environments. For example, adaptive systems should support many existing platforms, a source code of the application should not be required, inexperienced users should be able to compose systems depending on their environment, and performance degradation should be avoided. Therefore, FreeNA should be designed by taking into account these problems in order to achieve a general-purpose framework for adaptive communications and autonomous computing.

In this chapter, the fundamental policies for the proposed system are considered to represent a vision for our above-mentioned goals. We clarify the purposes of FreeNA first. Based on these purposes, the supposed users and usage scenarios are explained. Next, the characteristics that FreeNA should have are considered. Moreover, some network services examples are also given to show the usefulness of FreeNA.

3.1 Purposes

In Chapter 1, present issues concerned with the current network environments and/or systems are explained. It is difficult to restructure the core technology of current networks as they evolve and are diversify because a lot of systems depend on traditional

technologies. Many businesses are now depending on their existing network environments and more sophisticated network features are required such as secured and QoS-ensured communications or fault-tolerant network systems, which indicates that the network systems should be continually extended to fulfill such requirements while maintaining their compatibility with numerous traditional systems. The following are the change factors of the network environment, and network systems will need to be extended in response to these changes.

Performance Network performance is influenced by certain factors, such as bandwidth, protocol effectiveness and efficiency, network utilization, or computer performance. In particular, the network bandwidth and computer performance have been rapidly growing rapidly, and they are major influences on network performance. Bandwidth of current networks approximately range from tens of kbps to Gbps. Therefore, network systems should adapt the amount of data or transfer rates to take full advantage of the available bandwidth. Moreover, the buffer size of the network system can also be adjustable to enable for a burst transmission for the high utilization of network or long-term buffering for avoiding having to discard packets.

Quality of Service Quality of Service has become an important aspect of networks as because real-time communications or Service Level Agreements (SLA) are now required. In real-time communications, the packet stream should be continuously and stably transferred. However, there are interrupting many factors in a stable communication, such as packet discard, delay, and jitters. Generally, flow control or the intermediate routers or switches use a priority queuing method to ensure the intended packet stream. On the other hand, the QoS mechanisms of end systems have also been proposed, such as the application-level bandwidth shaper[24] and congestion control method[58] over the UDP protocol.

Security Security concern is an imperative issue for networks today. As new services are deployed on a network, many threads are found and used to attack the network. Consequently, a lot of security mechanisms are continuously evolving. Typical security mechanisms are data encryption, access control, and anti-malware technologies. An encryption algorithm is a common security method for user applications, and it is possible to adaptively select the necessary encryption algorithm depending on the required level of security. Moreover, other methods are also adaptable by constructing a sandbox environment for the application and introspecting the received application data.

Reliability A reliable protocol can be used to ensure the reliability of communications. Fundamentally, many networks are based on IP protocols that only offer a best-effort service, and upper-layer protocols like TCP should ensure a reliable communication. In addition to the TCP, there are other transport-layer protocols such as SCTP[59] and DCCP[60]. Moreover, session migration technology is also useful for preventing the influence of a system failure to existing communications.

Scalability Network scalability and the number of service users are larger now than ever before, and therefore, server systems should be designed to efficiently deal with numerous client systems. Although upgrading system components is an easy way, it cannot utilize these components effectively unless the system software and protocols are scalable. In practice, the application's processing algorithm can be parallelized, the server system can be replicated, and the caching mechanisms can be introduced to decentralize the processing cost and resource usage.

Diversity As the network environment has matured, a variety of network nodes are now connected to the network via various communication media. Since the characteristics of communications are vary with the network terminal types and media, network systems need to adapt to these variations to ensure there are efficient communications. For example, packet losses from low quality wireless channels are normally treated as

network congestion by the TCP protocol, which results in performance degradation. So, another wireless-ready transport protocol should be used when the user is in a wireless environment.

Mobility Network mobility has gathered a lot of attention as an important technology for ubiquitous computing. In mobile networks, the network nodes generally have a lower machine performance, and are dynamically moved to another network. Thus, the communication flow itself should migrate with the application, and a low-load communication mechanism should be adopted. In addition, reducing the amount of communication data or processing time would reduce the amount of power consumption.

Attempts need to be made to enhance existing systems to meet the network environment of advanced network systems. Naturally, users can introduce advanced systems while old systems are discarded, or directly update existing systems. However, these ways require an enormous workload and cost a lot of money. In particular, important systems including commercial systems cannot be directly upgraded without careful operational testing. Therefore, a specialized mechanism is needed in order to make existing network systems rapidly adaptable to environmental changes saving the effort needed for system modification and its cost.

This study is an examination of the user-oriented network service framework for existing network systems named FreeNA. To achieve the above goals, FreeNA has to be designed as follows.

- **A variety of network services are available with FreeNA from the lower-layer protocols to upper-layer protocols.**
- **Users can flexibly compose independent network services, and adjust their behaviors with specified parameters**
- **FreeNA can work with existing network environment applications without any modification of the application, OS, and protocol specifications.**

- **Not only professional developers but also non-developers including application users, system administrators, and network administrators can leverage FreeNA to use advanced network services.**

3.2 Supposed Users and Usage Scenarios

As described in the previous section, non-developers also can use FreeNA because they use their applications without needing advanced expertise. However, users have to know what approaches are most useful for their systems even though they do not need to understand the technical content of the approaches. Therefore, FreeNA is designed for application users with a certain level of technical knowledge, developers, administrators, and researchers. They may introduce new features into their existing systems, or customize behaviors to fit the surrounding network environments. However, these processes will require not only the source code of the applications and development environment, but also their modification. Then, FreeNA enables users to transparently insert additional functions into their systems without the source codes and any modification.

Developers Developers can utilize FreeNA during development for two reasons. The first reason is that developers only implement the core functions of the application by incorporating the adaptivity mechanism as a structurally different independent component. Realistically, a variety of network services should be supported for future environmental-change and required services reasons that cannot be determined during development. Although it is beneficial to design an application to be easily extended after completing the development phase, developing flexibly extensible applications requires complicated structures in many cases. Therefore, FreeNA can be used to provide adaptivity into an application while maintaining a simple structure. The other reason is that FreeNA enables developers to independently examine network services in actual environments from the core logic of the application. This can prevent unintended system modifications and associated program errors. This usage is also useful when developers can only partially access source codes such as the distributed development, because developers can compose

arbitrary stub components instantly for early prototyping.

Administrators System/network administrators can manage the network system rather than developers. Administrators always have to monitor their systems and transfer data, deal with sudden trouble, and upgrade systems in response to system usage changes. Therefore, even during an operational period, network systems should be flexible enough for the administrators to dynamically add various functions into running systems. FreeNA facilitates such administrator tasks using utility services such as data introspection, flow control, and arbitrary protocol composition. In fact, many network tools already exist that have the above-mentioned functions. However, FreeNA is different from these tools in that administrators can use the services on a common platform with a unified usage.

Researchers Network researchers have to examine a variety of network features such as the network architectures, protocols, and traffic data for their research. These studies usually require the combination of a lot of parameters and data patterns. Therefore, there should be a dedicated framework that can compose networking functions using arbitrary parameters, and measure their characteristics. In practice, researchers are using sophisticated network simulators like OPNET[61] or NS2[62] to evaluate intended network models. Nevertheless, these simulators can forge all the different networking functions and data patterns, but researchers still have to evaluate their models in a physical environment. This is because practical performance and detailed behaviors also depend on the physical network architecture, hardware components, implementation of the protocol stack and applications, and real data traffic. FreeNA allows researchers to construct supposed network systems and evaluate them without significantly influencing existing systems.

Application users Generally speaking, mere application users use their applications as they are, and they do not need to change them. In some cases, the application cannot work well in certain user environments, and the application has not been carefully maintenanced. For example, a user wants to watch an online movie on an express train,

and the user can tolerate low picture quality, but cannot accept picture and sound interruptions. In such cases, the users can decrease the picture quality by transforming the video codec and appending an error correction mechanism to the data stream using FreeNA. Even though users do not know the technical details, they can adjust the application's functionalities if they know the relationship between the intended functionality and corresponding service component name.

3.3 Characteristics

The purposes of FreeNA and its usage have been explained. One major characteristic of FreeNA is its general-purposed platform for various types of users, rather than being a specialized tool for experienced developers. Therefore, FreeNA must be designed taking into consideration not only its functionality but also its usability or compatibility. To achieve the goals set for FreeNA, it is designed to providing the following characteristics.

Implementation Form There are many types of implementation forms for realizing a functionality, such as API, middleware, OS, virtual machine, or proxy server. When you carefully consider the supposed users of FreeNA, FreeNA should not be in the API-form because the users have to write a program to call FreeNA's API for the service composition. That is, FreeNA should be used as an independent system such that the users can use it without writing any type of driver program. The next consideration is that FreeNA should be used in current widespread systems. Consequently, the current form of operating system is not suitable for FreeNA because commonly used operating systems cannot be replaced with another one. In addition, FreeNA should be used as only a compositional adaptivity mechanism, rather than to support a fully completed functionality like hardware management, memory management, or fundamental network processing. Therefore, in this study, an executable middleware implementation form is adopted for FreeNA because the basic functionality is provided by the underlying OS, and users can attach FreeNA to the intended application in order to flexibly control the application's behavior.

Usage In general, many similar systems suppose that the system users and network service developers are one in the same, and technical knowledge and the appropriate skills are required to use these tools. As a result, it is possible for users directly write an appropriate glue program between the application and the network service component. In contrast, since FreeNA is designed for inexperienced users too, its usage needs to be carefully considered. To create a user-oriented system, the composition of the incorporating network services and operation of FreeNA itself have to be set by the users without them incurring any difficulties. For this reason, FreeNA offers configuration-based composability and command-based operability to its users. Users can describe the names of the intended network services in a readable configuration file for service composition. Then, FreeNA reads the file and carries out the incorporating specified services by using operational user commands. It will be possible for even non-experienced users to easily adapt their applications to the required environment as long as the implementations of the network services are already available.

Network Service Implementation As previously described, it is better to implement a network service as an independent component because users can insert the service component into the application in a structurally-separated manner. Multiple network services can also be composed together based on the user's directions because they do not have to depend on other network services. In addition, users can obtain network service implementations from third parties. If there are a lot of these types of service distributors, this process will be even more convenient for many users because they will not need to implement the intended functionality on their own.

Platforms To deploy FreeNA into existing network environments, it must be designed to work on the multiple platforms currently being used. In general, it is almost impossible to develop a portable system that is compatible with a lot of platforms. However, many platforms currently provide similar network interfaces like the BSD socket interface to applications including UNIX variant systems and Microsoft Windows, and dynamic mod-

ification mechanisms. Consequently, FreeNA can be developed as a portable framework by using the socket interface and adaptive techniques. In this paper, I describe a FreeNA I developed for Windows and Linux operating systems, which are the dominant platforms in the actual environment.

Applications Even though almost all applications are use socket interfaces for network functionality, applications themselves are implemented in different programming languages. Naturally, FreeNA should not depend on the applications implemented in particular programming language. Fortunately, common operating systems provide socket interfaces as system-calls or library functions, and FreeNA can interpose in the socket-call's behavior at the binary level. As described in chapter 2, several adaptive techniques are available to modify an application's behavior besides the language-based approach.

3.4 Network Service Examples

FreeNA is designed as a general-purposed framework such that a variety of network functionalities can be composed flexibly at a user's will. In this section, the assumed network services incorporated by FreeNA are introduced.

- **Compression**

  A compression service shrinks the data being sent before passing it along to the underlying operating system on the sender side, and stretches the received data before passing it along to the application on the receiver side.

- **SSL**

  The SSL service provides a secure network communication to non-SSL-compliant applications on the PKI platform by overriding the existing TCP connections. Compared with SSL-proxy systems like Stunnel, the proposed service directly connects both end systems with an end-to-end transparency and better performance.

- **Error protection**

  The error protection service protects the end-to-end communications by hiding

the packet error from the application. Some examples of this in practice are the forward error correction (FEC) codes like the Reed-Solomon code[63] mechanisms or the packet duplication methods[64] to the packet stream.

- **TCP multiplexing**

  The TCP multiplexing service brings multiple TCP connections together into one connection. This is effective if there is an already existing TCP connection, because the actual throughput will be increased by using the same connection instead of establishing a new one.

- **Ad-hoc traffic shaping**

  The ad-hoc traffic shaping service adjusts the transmission rate of data packets in the user-space or controls the behavior of the TCP protocol by setting the socket parameters like TCP_MAXSEG. Therefore, users can control the bandwidth utilization in an ad-hoc way without needing special tools like Trickle.

- **Stateful application-layer firewall**

  The stateful application-layer firewall is different from common packet filtering based firewalls in that this service checks the packet content or aggregated content to inspect the application-layer protocols. This service will be effective for preventing applications from application-specialized attacks, such as buffer-overflow attacks, format string attacks, and SQL injection attacks like that of TCP Stream Filtering. Moreover, it will enhance the security of the P2P node–P2P communications like file sharing, which has now become the major hotbed for viral infection by inspecting suspicious content during the communication time.

- **Mobility**

  Mobility services manage the connectivity of the end-to-end applications by migrating the application-level sessions like TESLA. Moreover, the power consumption of mobile terminals can be controlled by reducing the transmission of packets like MetaSockets.

- **Non TCP/UDP transport-layer protocols**

  Normally, applications can use the transport-layer protocols offered by the underlying operating systems, and in most cases, TCP or UDP is used. However, many researchers have pointed out that these protocols are not optimal in some network environments, such as wireless networks, congested networks, and long fat pipe networks. Therefore, this service allows for additional transport-layer protocols like SCTP and DCCP to be transparently installed in existing systems.

In practice, there are dedicated systems that offer one of the above network services, such as Stunnel for SSL and compression, DummyNet[65] and Trickle for traffic controlling, and TCP Stream Filtering or Zorp[66] for application-layer firewalls. The users can select these tools instead of FreeNA. In particular, users may have no choice but to use dedicated tools for lower-layer services because FreeNA only limitedly supports such services.

However, there are still benefits to using FreeNA, for example, users can use many network services in an unified way on multiple platforms, that is a customized service for the intended application. Therefore, the major advantage of FreeNA is that it provides a mechanism to users to more easily customize and combine services without needing special platform and application support. In other words, FreeNA is a kind of application/user-oriented network tool when compared with other networking tools.

Note that, although FreeNA can support a variety of network services and applications, there are cases where some services are not practical for some types of applications. As a result, users have to consider whether the supposed services are effective enough for their applications.

3.5 Summary

In this chapter, the fundamental policies for FreeNA were discussed for creating a general-purpose framework for adaptive communications and autonomous computing. Many practical networks are currently required to offer more sophisticated network fea-

tures such as secured and QoS-ensured communications or fault-tolerant network capabilities. Network systems will need to be continually extended to fulfill such requirements while remaining compatible with numerous traditional systems.

In this study, FreeNA is designed as a common platform for variety of network services, flexible system that users can configure its functionality, portable system for multi-platforms, and user-oriented system supposing non-experienced users. Therefore, implementation form, usage, network service implementation, platforms, and applications must also be deliberated unlike other similar systems.

Compression, SSL, error protection, TCP multiplexing, ad-hoc traffic shaping, stateful application-layer firewall, mobility, and additional transport-layer protocols services are examples of the network services that are available on FreeNA. Even though there already are many networking tools that provide these services, there are benefits to using FreeNA in that it is a kind of application/user-oriented network tool providing a mechanism to users for customizing and combining services in an easier way without needing special platform and application support.

Chapter 4

Architectural Design and User Experience

In the previous chapter, the fundamental policies were clarified for constructing a FreeNA system. Compared with other similar systems, the major characteristics of FreeNA systems are that it enables composable network services for existing systems with higher-level user instructions. So, FreeNA is designed to be implemented as an independent middleware taking into account the possible implementation properties.

In this chapter, the architectural design of FreeNA as a middleware system is explained in light of FreeNA's characteristics. Since FreeNA is composed of several subsystems, the role and relationships to the other components are clarified for each subsystem. In addition, actual usage of FreeNA with the configuration file for a service composition and operational commands is also described to show the usability of FreeNA.

4.1 Network Services Perspective

4.1.1 Service Insertion Mechanism

As has been described, FreeNA enables its users to transparently add arbitrary network services to existing applications. In practice, network services are inserted into the communication path between end-to-end applications as pictured in Fig.4.1. Generally, the application transmits the data by invoking the corresponding socket-calls. At this point, the data is already manipulated by the upper-layer protocols like SSL and RTP not just the application-layer protocols. Then, the data is passed to the socket interface

within the kernel. Inside the kernel, the data is encapsulated as packets or frames by the lower-layer protocol headers. Finally, these packets are transmitted to the network as a physical signal by the network card. On the receiver side, the received packets are passed to the application in reverse order.

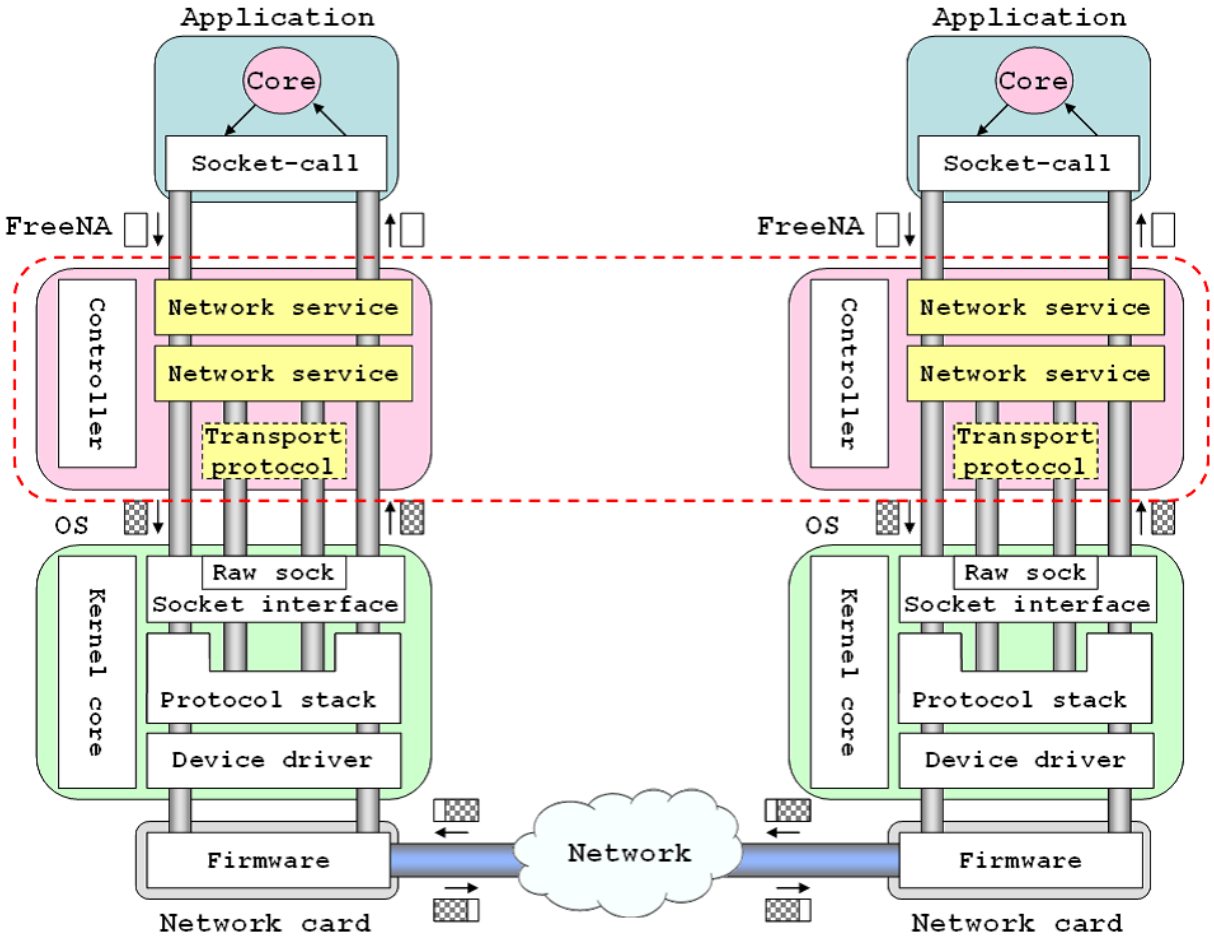

Figure 4.1: Perspective view of network service insertion

On the other hand, when FreeNA is used, additional data manipulation mechanisms are introduced between the application and socket interface within the kernel. The network services are implemented as independent program libraries, and they provide several functions that have the same interface as the socket-functions. Particular socket-calls are intercepted by FreeNA, and linearly-linked network service functions are executed instead of executing the internal socket functions. During transmission, the data passed by the application is manipulated by the inserted network service functions. For example, the

data is encrypted by *an encryption network service* whenever the application invokes a socket-call for sending. After that the encrypted data is passed to the underlying operating system by the intercepted socket-call. In contrast, the received data is taken from the operating system by intercepting the socket-calls for receiving, and then the data is properly manipulated by the inserted network services. Take the same example again, the received data remains encrypted, and therefore, is decrypted by *a decryption network service* before being passed to the receiver application. Not only the upper-layer services, but also the transport-layer services can be inserted. In that case, the traditional transport-layer protocol mechanism within the kernel is deactivated, and the network-layer protocol mechanism is offered directly to the inserted service. Naturally, the transport-layer services must be located as the bottom-most inserted network service, and only one transport-layer service can be used at a time.

Note that inserted network services are completely hidden from both end-to-end applications because network service functions behaves like corresponding socket-calls. In other words, applications can enhance their networking functionalities implicitly by just invoking socket-calls as usual.

4.1.2 Flow Handler Structure

When inserting network services, FreeNA uses a notion of the *flow handler*[23]. Basically, the flow handler takes one input data flow and multiple output data flows. In this research, each network service corresponds to a flow handler instance, and FreeNA combines one output flow of the upstream handler with the input flow of the downstream handler in order to achieve the linearly-linked structure of the network services.

Figure 4.2 depicts the fundamental structure of the flow handler in this research. As described before, each network service is implemented as an independent program library that has socket-like functions. Therefore, calling these functions can be corresponded to the input flow of the flow handler, and the input data is passed to the library function as an argument. The input data is manipulated inside the function as the corresponding protocol processing. In a normal function call, the manipulated data will be returned to

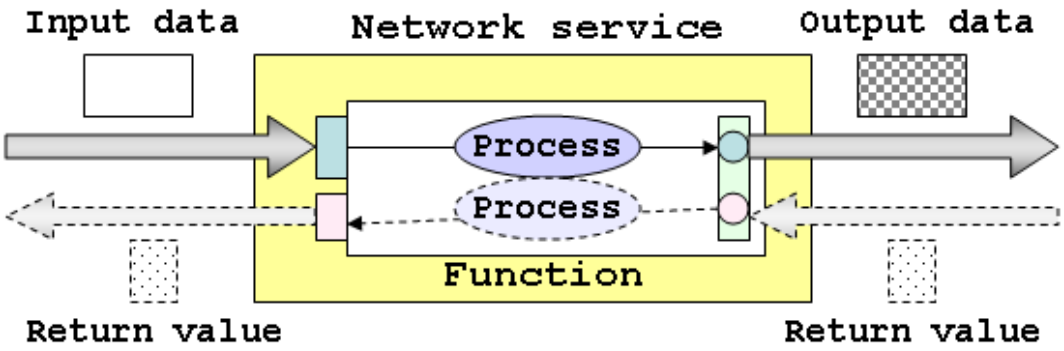

Figure 4.2: Flow handler structure for network services

the caller as a return value of the function. However, in order to linearly concatenate multiple flow handlers, the manipulated data has to be passed to the downstream library via another function call. Therefore, the corresponding function of the downstream library is called within the currently executing function to output the manipulated data. Likewise, the return value from the downstream library's function is passed to the upstream library when the current function is finished.

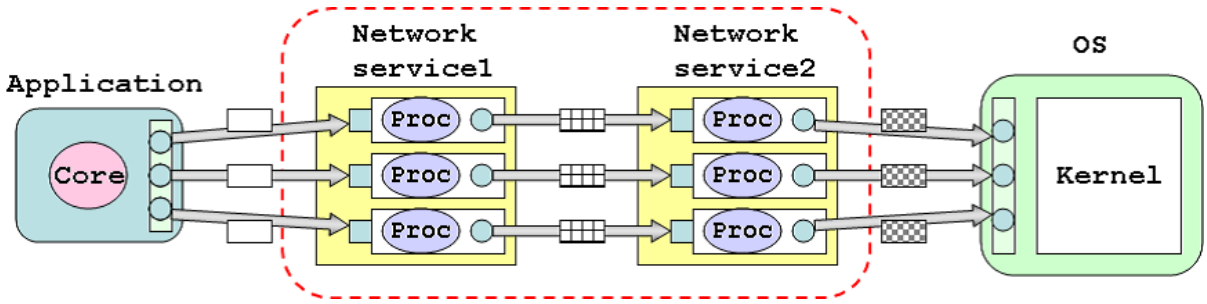

Figure 4.3: Flow handler chain between the application and OS

In practical usage, several flow handlers are concatenated in line (*flow handler chain*), and interposed between the application and OS. Figure 4.3 shows the perspective view of the flow handler chain structure. In the figure, two network services are inserted between the application and operating system. These two services execute different manipulation to the input data, for example, the first service encrypts the data, and then, the second service compresses the encrypted data. The actual socket-call process is as follows. First, the application calls a socket function like `send()` with the data to be transferred. Instead

of executing the authentic socket function, the corresponding function of the uppermost network service library is invoked, and the data is passed as one of the arguments. Within the network services, the input data is manipulated and passed to the downstream network service one after another. Finally, the lowermost library function calls the authentic socket function within the kernel.

Note that, since the corresponding socket call varies from the intended function to the actual function, each network service provides multiple functions. As a result, multiple data flows (ex. transmitting and receiving) are used at the actual time, and these data flows are independently treated by the network services.

4.2 FreeNA Architecture

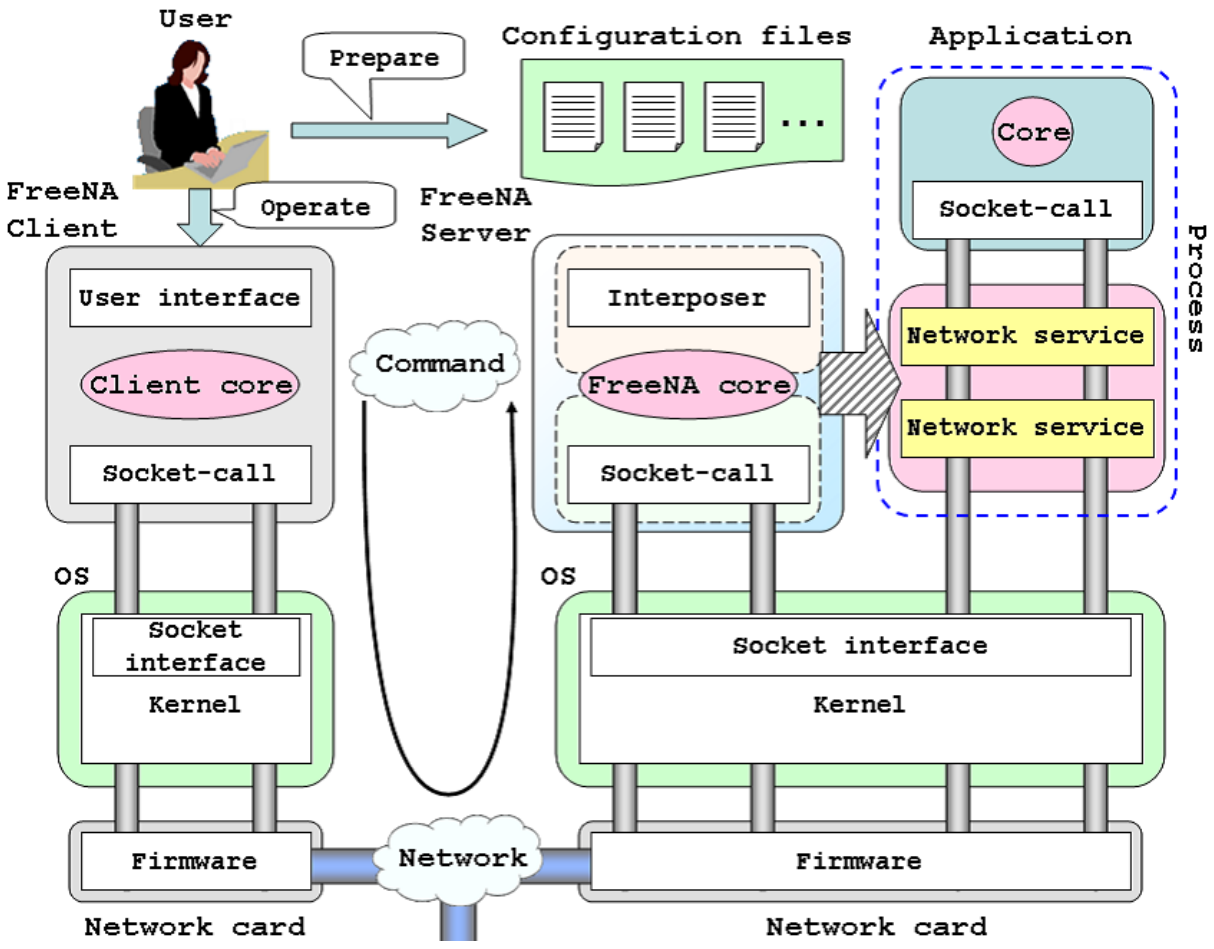

Figure 4.4: Overall FreeNA architecture

In this section, the fundamental architectural design of FreeNA and its usage are explained. Figure 4.4 shows the overall architecture of FreeNA. As illustrated in the figure, many components are complicatedly involved in constructing a FreeNA system. The main FreeNA system can be divided into the *FreeNA client* and *the FreeNA server*. The FreeNA client subsystem is used to provide a mechanism for controlling the FreeNA system, and the FreeNA server executes the core functionality of FreeNA, such as the composition and insertion of network services. Strictly speaking, both subsystems are implemented as executable user-programs, and they only perform utility functions rather than the network services themselves. Instead, the network services are embedded into the application process and executed directly at runtime. Since FreeNA consists of complex components and technologies, the internal architecture and role of each part of FreeNA is explained in detail in the following sections.

4.2.1 The FreeNA Client

The FreeNA client is a normal executable program that provides a user-interface mechanism to FreeNA users. Such a dedicated operational system is needed because FreeNA uses specialized techniques a lot for creating transparent network service compositions and insertions on multi-platforms. That is, the FreeNA client absorbs the complexity of FreeNA by providing user-friendly operational commands to its users. The following is a fundamental capability of the FreeNA client.

- **Command input**

  Users can input operational commands using terminal emulators or other graphical interfaces like a Web browser. Command syntax and arguments are also checked. Multiple commands written in a batch file can also be input at one time.

- **Standard-input**

  Users can specify standard-input data for the application in the same way as the command input. The input value is directly passed to the FreeNA server.

- **Standard/Error-output**

  The output message from the application can be displayed to users.

- **Server control**

  Users can control the FreeNA server via the client even if the users are on different machines than the server. At first access, the client can be authenticated by the server.

Table 4.1: List of major commands of the FreeNA client

Command	Description
run	Execute the specified application
dumpfile	Create the executable file
stop	Suspend the specified application
continue	Reexecute the specified application
terminate	Terminate the specified application
detach	Detach the specified application
input	Input standard-input data to the application
proclist	Show a list of running application
cd	Change the current directory
ls	List all files in the current directory
pwd	Show the path to the current directory

Table 4.1 is a list of representative user-operational commands offered by the FreeNA client. Users can input these commands on their terminal emulators as shown in Fig.4.5. As you can see, users can easily and agilely use the FreeNA system without special knowledge of the technical mechanism. Note that the current user-interface is a command-based interface like a GNU debugger (GDB) or a client system for a database management system (DBMS). However, another version of the FreeNA client containing a GUI interface like a browser-interface can be made possible because the core functionality of the FreeNA system is implemented within the FreeNA server, and an internal protocol architecture between the client and the server is also defined.

Figure 4.6 depicts an internal architecture of the FreeNA client. Regardless of the type of interface, each user-operation is processed as an internal command. The main steps for the command processing are as follows.

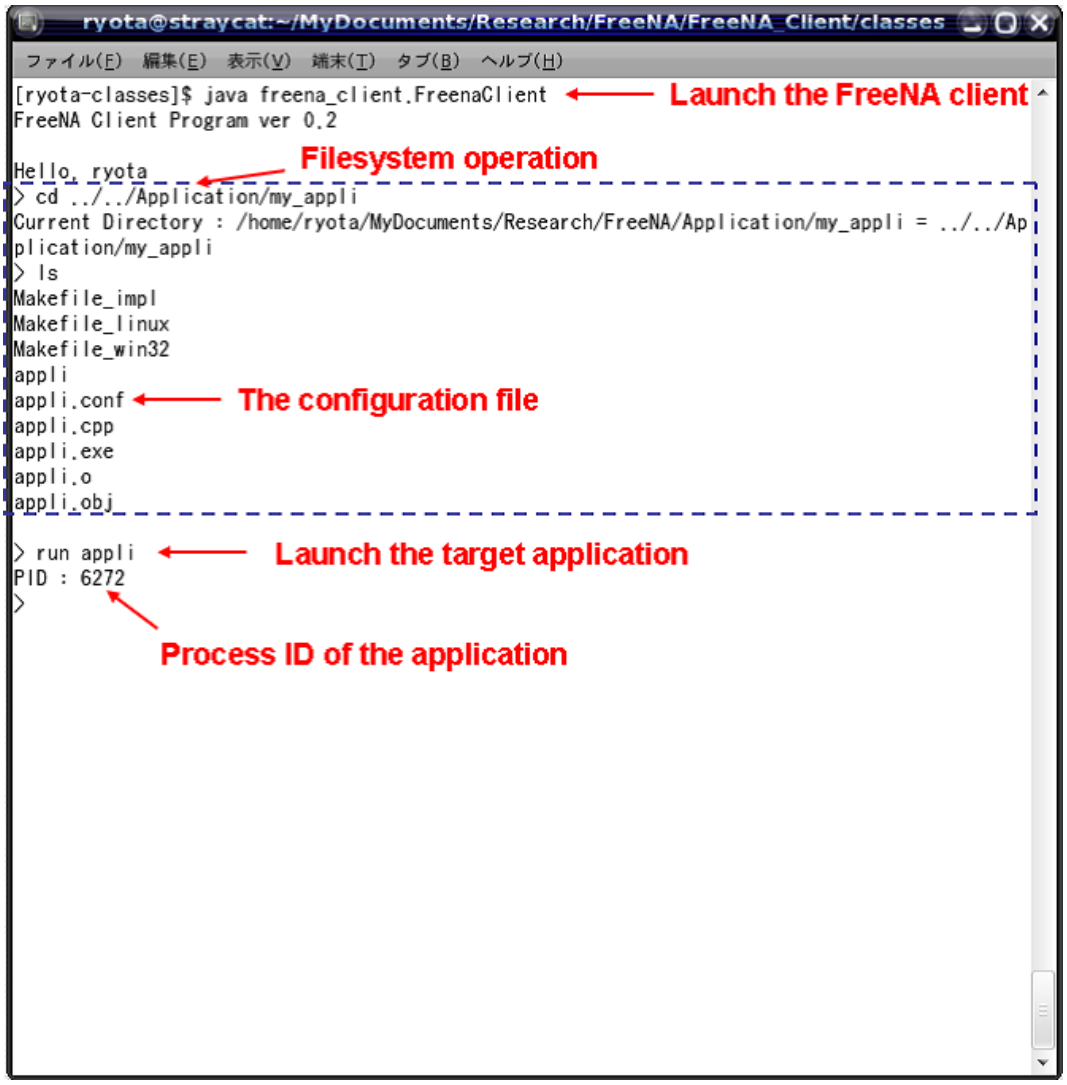

Figure 4.5: Example of FreeNA client usage

1. An operation command is input by the user using the terminal emulator or browser.

2. The input command is parsed to validate the command format and extracts a corresponding command type.

3. The command dispatcher dispatches the appropriate task based on the command type.

4. If a server's process like the service insertion is required, the command and its arguments are encapsulated by the protocol composer to be transferred to the FreeNA server.

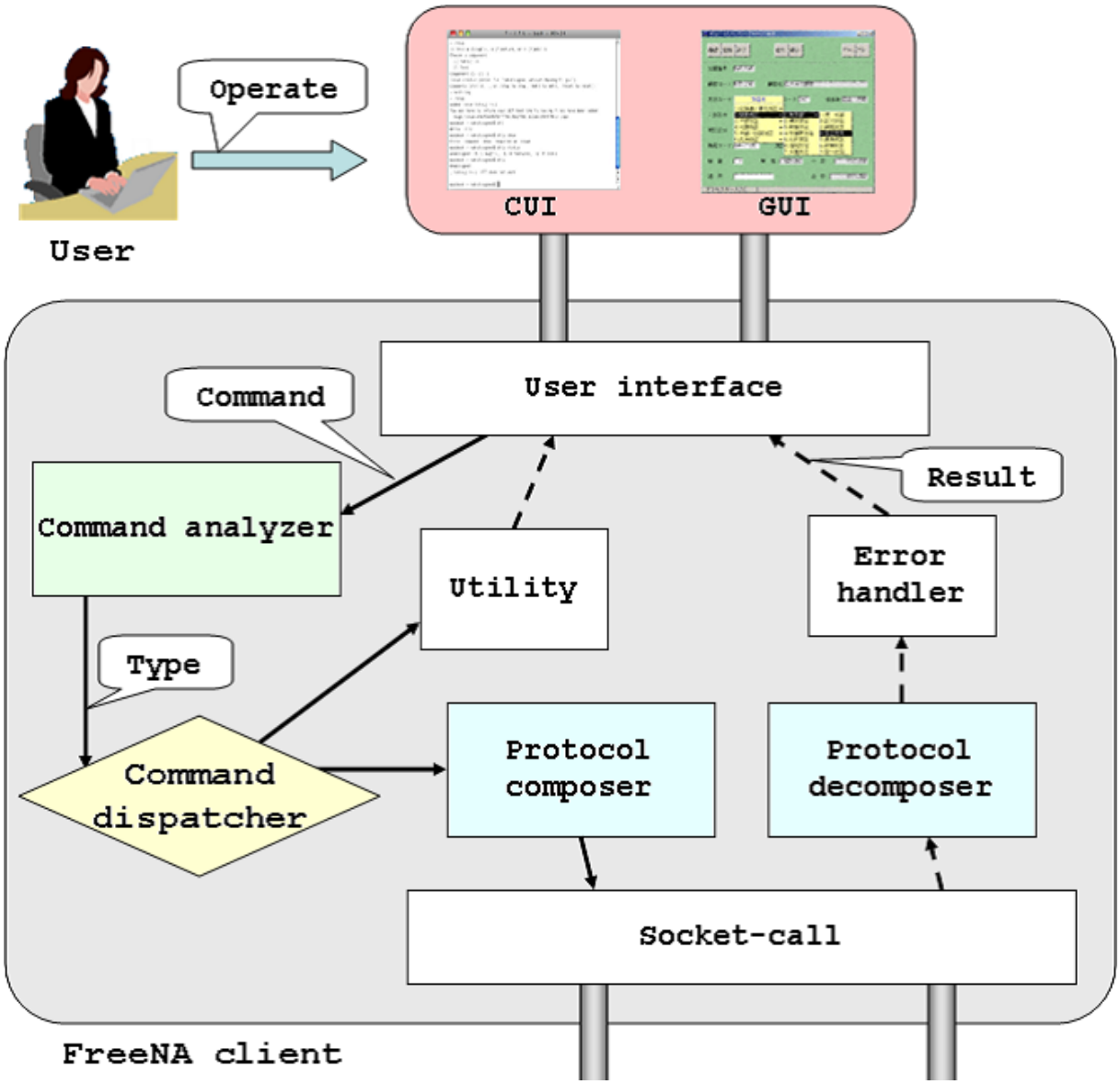

Figure 4.6: Perspective architecture of the FreeNA client

5. A protocol decomposer receives the data from the server, and extracts the returned message.

6. The returned message is displayed to the user.

In this way, the input user-command is a trigger to operate the FreeNA system. In total, the FreeNA system works as an event-driven system based on the internal commands. Each internal command basically corresponds to one of user-operational commands.

4.2.2 The FreeNA Server

The FreeNA server is in charge of the core processes of the FreeNA system, such as the invocation of the target application and inserts the specified network services based on a configuration file. The following is a synopsis of the capabilities of the server.

- **Application launch**

  User-specified applications can be launched with command-line parameters and redirection settings. Technically, the application process is created as a child process of the FreeNA server. The network services are inserted when the application process is started.

- **Application attachment**

  Already running applications can also be attached in order to insert network services. Note that the consistency of the communication data before and after service insertion is not automatically ensured.

- **Application termination**

  FreeNA-attached applications can be forcibly terminated by specifying the process id.

- **Network service composition**

  Specified network services can be composed as the flow handler chain structure for each socket function.

- **Network service insertion**

  Composed network services can be inserted into the application by intercepting the socket-calls from the application.

- **Configuration parsing**

  Information from the intended network services and their compositions can be extracted from the configuration file.

- **Process management**

  FreeNA-attached application processes are managed by the process IDs for standard-input/output of the application.

- **User management**

  The FreeNA server provides a login facility and access control list mechanism to each user.

- **Directory operation**

  Users can traverse the directory tree using operational commands.

Like the FreeNA client, the FreeNA server is implemented as an independent executable program. But the FreeNA acts as a central system that executes various requests from the client systems. That is, the FreeNA server does not provide user-interface functions, and is controlled via the client systems. This separation of functionality is effective from the following aspects. First, the FreeNA server and target applications can be managed by administrators in an integrated fashion because general users do not directly need these systems. This will be useful for constructing thin-client environments, or enforcing users to use the applications with specified network services. Next, the core components of the FreeNA system can be implemented as portable systems because their user-interface functions depend on the users' platforms and environments. Moreover, it is possible for the client and server systems to work in different execution modes. Since the FreeNA server should be able to sufficiently use the platform functions, the FreeNA server has to work in the privileged mode. On the other hand, the FreeNA client can run in the user mode to prevent users from being given a privileged right.

Figure 4.7 depicts the internal architecture of the FreeNA server. The FreeNA server takes on the transaction method, and each server's process is triggered by the request command from the FreeNA client. Internally, the structure of the FreeNA server can be divided into two parts, a platform-independent part and a platform-dependent part, to ensure the system's portability. Most functions of the server are implemented in the

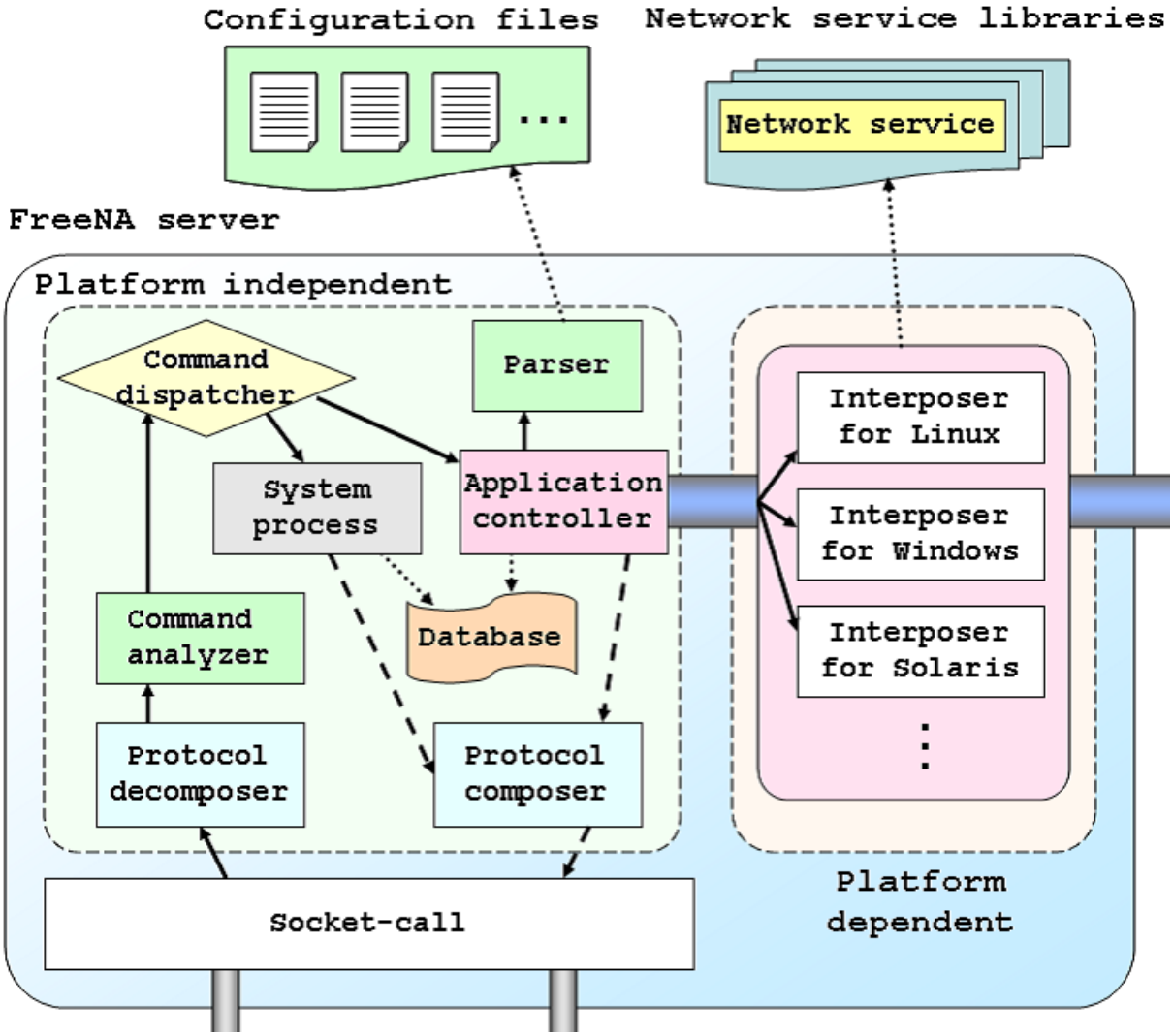

Figure 4.7: Perspective architecture of the FreeNA server

platform-independent part, such as the command processing, network processing, configuration parsing, and network service composition. On the other hand, the application launch and network service insertion are performed in the platform-dependent part because these processes require some process manipulation techniques. Practical manipulation techniques will be described in a following chapter. The main steps of launching the application and service insertion are as follows.

1. A request command and arguments are received from the FreeNA client in a dedicated protocol format.

2. The protocol decomposer is used to decompose the received data, and the command

type is identified by the command analyzer.

3. If an application execution is requested, the configuration file, which is one of the arguments, is parsed by the parser to determine the network services to be inserted, their composition, and service parameters.

4. The network service composition such as the service order and socket-calls to be intercepted is calculated by using the application controller.

5. The interposer program corresponding to the underlying platform is invoked.

6. The target application is launched as a child process, and at the same time the required network services are appended to the process by the interposer.

7. The process ID of the application process is registered to the internal database, and transmitted to the FreeNA client with a message concerning the success of the request.

The FreeNA server performs the network service composition and insertion by interlocking the platform-independent and platform-dependent parts in this way. Since application manipulation techniques vary from platform to platform, several types of interposer programs can be prepared for use as the platform-dependent part. Note that there are other server components not described above. The system process performs the user management and directory operations. The user information, like the login name and access control list, and running process information are registered to the internal database.

4.3 Conceptual Approach to Ensure end-to-end consistency

In this section, the connectivity and consistency of the network services between end-to-end applications are discussed. As described before, FreeNA determines the network

services to be statically inserted based on the configuration file. That is, both end users must cooperate with each other so that they configure the same service composition with the same parameters into the file in advance. However, users of large-scaled server systems that communicate with a lot of clients have difficulty statically configuring with each client system. Therefore, FreeNA should provide a mechanism that ensures there is an end-to-end consistency in the network services without static configuration.

4.3.1 Dynamic Service Composition based on Negotiation

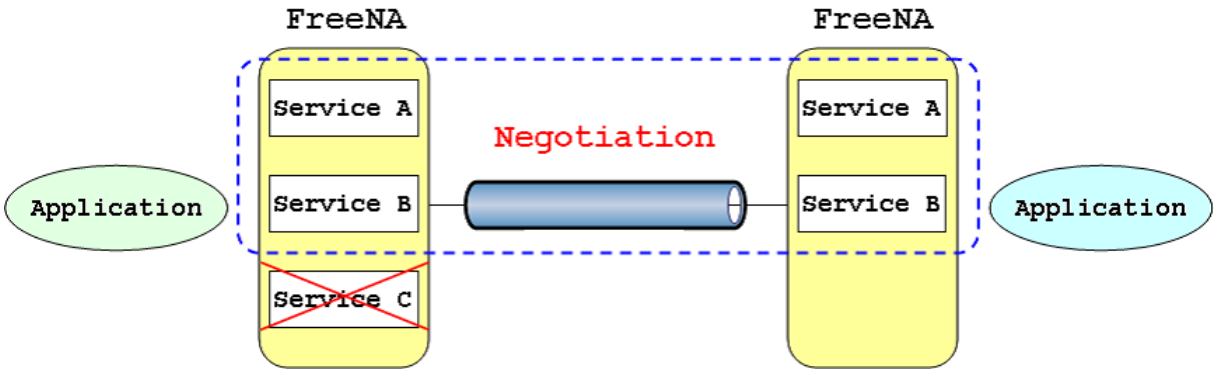

Figure 4.8: The overview of the negotiation for dynamic service composition

To dynamically compose appropriate network services, a negotiation mechanism is introduced into FreeNA. By negotiating the required services between end-to-end FreeNAs before starting an application-level communication, the end-to-end applications can work under the same network service configuration (Fig. 4.8). Considering there are so many network services and parameter settings, it is a desirable approach that users can offer several candidates for services and the parameter settings so that FreeNA can automatically decide on the required services from the candidates during the negotiation. This configuration mechanism is explained in section 4.4. The important point is that FreeNA should take into consideration the remote applications that do not run under the FreeNA environment in order to transparently deploy the FreeNA system into existing wide-spread network environments. For this reason, FreeNA should check whether the remote application is running with FreeNA before the negotiation. The details of the mechanism for the

determination of FreeNA's existence, and the practical negotiation protocol are described in section 5.3.

4.3.2 Transport-Layer Protocol-Free Environment

Some network services should work with FreeNA because the service usage under the network environment is not defined even though the service algorithm itself is standardized. For instance, the AES encryption algorithm is standardized by the National Institute of Standards and Technology (NIST)[67], however, its usage on the networks varies from IPsec[68], SSL/TLS[69], or WPA2[70]. When AES is directly applied to the application messages as ad-hoc encryption, its usage (key-length, encryption mode) should be decided by FreeNA to ensure consistency. Therefore, general-purposed network services should be composed by FreeNA's negotiation scheme.

On the other hand, transport-layer protocols have standardized functions and usage. So, the FreeNA-enabled application that uses an inserted-version of a protocol can easily communicate with the traditional application that uses a kernel-provided-version of the protocol. In this study, the transport-layer protocol to be used can be decided without a negotiation for the upper-layer network services. As a result, the FreeNA-enabled application can communicate with both the FreeNA-enabled application and the normal application using the same transport-layer protocol. The details of this mechanism (transport-layer protocol-free environment) are explained in section 5.4.4.

4.4 Configuration Mechanism

Although users can operate the FreeNA system by using the dedicated client system, it is inconvenient for users to have to configure the network services whenever executing applications. Therefore, users are allowed to create the configuration file for each application. The configuration file should be written without needing extra programming and the internal architecture of FreeNA taking into account that considering inexperienced users also use FreeNA. In this study, the XML-based format is adopted for the configuration file, and the functional selection approach is taken to describe the network services and

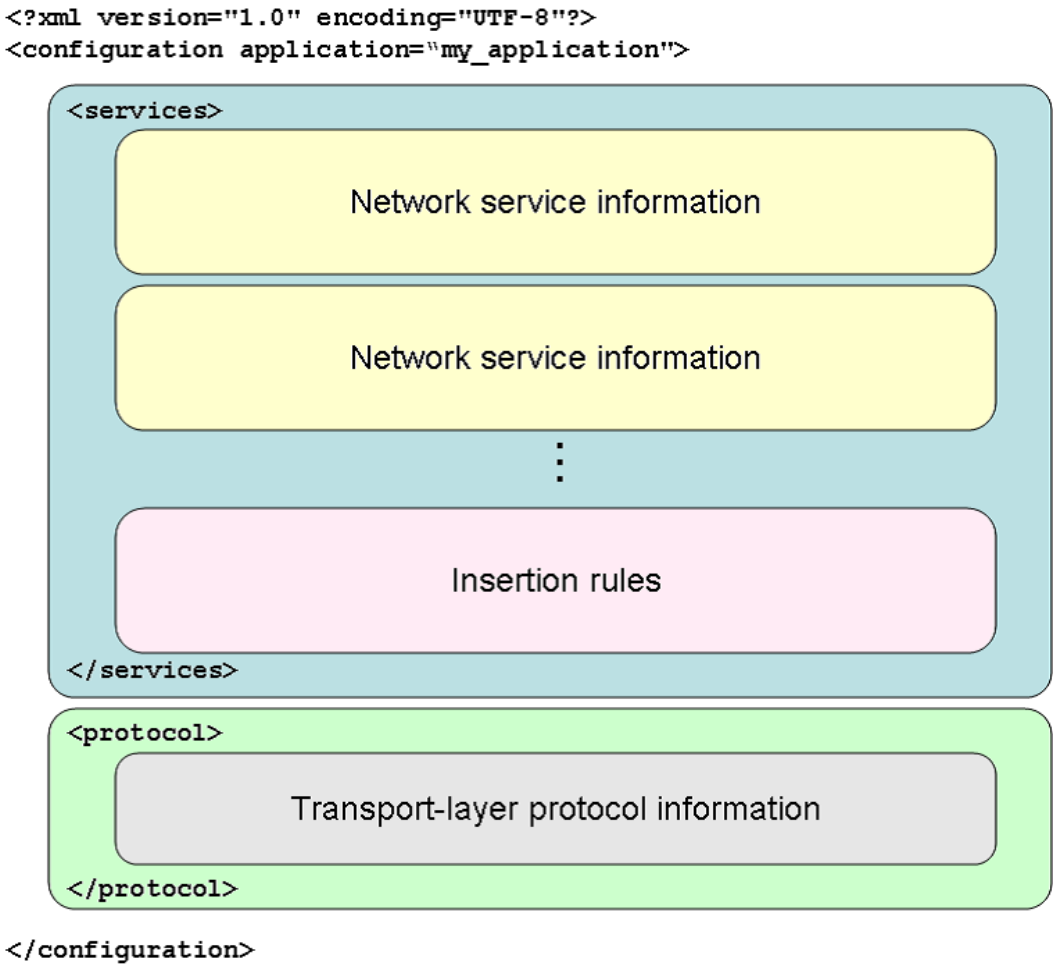

Figure 4.9: The outline structure of the configuration file

additional information.

To support the flexible customization feature of the application, the configuration file has variety of items (elements and attributes in XML-style) available to users. Figure 4.9 shows an outline structure of the configuration file. The configuration file mainly consists of two elements, a **services** element and a **protocol** element. The **services** element consists of two parts. The information concerning the network service to be inserted is described in the network service insertion part. An insertion rule part is used to set the condition of whether the specified network services are to be inserted or not. The **protocol** element is used to specify the alternative transport-layer protocol for traditional TCP/UDP.

4.4.1 Network Service Configuration

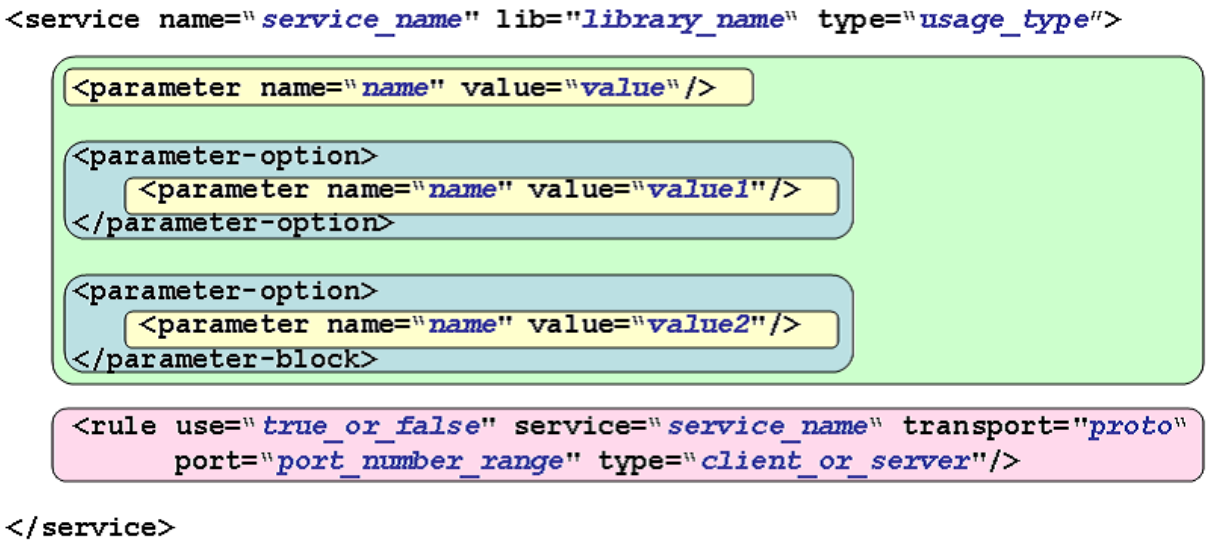

Figure 4.10: The structure of the service element

Information on each network service is described with a **service** element inside the **services** element. Figure 4.10 illustrates the structure of the **service** element. The practical syntax of the **service** element is described in A.1, and the important elements and attributes are explained below.

Identifying service

Each network service is identified by the **service** element by its **name** and **lib** attributes. Basically, the **name** is used for the user convenience, and the **lib** specifies the actual service library path.

Service type

The **type** attribute of the **service** element represents the service usage type during runtime as listed in Table 4.2. These service types influence the negotiation process. *Local* typed services are always applied to the application, but they are not seen from the remote. The logging service or the firewall service will be the local services. *Global* typed services can be used between the end-to-end applications synchronously, such as the compression service or the encryption service. It is better to compose these services

according to the negotiation, and the *global* services must be used during the application-level communication unless the *optional* condition is specified.

Table 4.2: Available options of service usage

Type	**Description**
local	Specified service is only applied at one end application locally
global(/required)	Specified service must be used at the communication
global/optional	Specified service can be omitted by the negotiation result

Service parameter

A **parameter** element is used to set the text-based parameters for the network service. The parameter can be specified using a parameter name and parameter value pair. The specified service library itself interprets the specified parameters at the time of execution.

Parameter option

A **parameter-option** element is used for FreeNA's negotiation to deal with cases when the service provides several parameter value candidates, but one of them must be used in the communication.

Insertion rule

A **rule** element (local rule) is used to specify the type of packet flow by using a set of conditions, such as the transport protocol, port number rage, and communication type. The specified service is only applied if the conditions are satisfied.

While the **rule** element within the **service** element (local rule) only works on the service, the **rule** elements of a **using-rule** element (global rule) can apply to all the network services written in the file. Figure 4.11 shows the structure of the **using-rule** element. Users can set multiple integrated conditions by writing multiple rules in descending prior-

```
<using-rules>
    <rule use="true_or_false" service="service_name" transport=
    "proto_name" port="port_number_range" type="client_or_server"/>
                                    :
</using-rules>
```

Figure 4.11: The structure of the using-rule element

ity order, and an '*' symbol can be used to express all of them (protocols, port numbers and so on). Note that the local rules come before the global rules

4.4.2 Protocol Configuration

```
<protocol>
    <default name="default_protocol" lib="library_name"/>
    <option name="optional_protocol" lib="library_name">
        <parameter name="name" value="value"/>
    </option>
</protocol>
```

Figure 4.12: The structure of the protocol element

Users can configure the transport-layer protocol used by the application. To create the transport-layer protocol-free environment, which is an automatic mechanism for dynamically determining the proper protocol. Therefore, the configuration file has two types of protocol settings, which are shown in Figure 4.12. A **default** element expresses the transport-layer protocol used by the client-type applications at the time of access. On the other hand, the server-type applications select the appropriate protocol written in one of the **option** elements to fit the client protocol. If there is no option protocol that fits the client protocol, the client access is denied by the server. In this way, users can create a transport-layer protocol-free environment.

4.5 Summary

In this chapter, the architectural design of FreeNA and the user experience were explained. First, the fundamental mechanism of a network service insertion was described. The inserted network services are located in the communication path between end-to-end applications by intercepting particular socket-calls by using FreeNA. Next, the concept of the flow handler and flow handler chain was introduced for independently composing each of the network service components. Each network service corresponds to an instance of the flow handler, and FreeNA combines one output flow of the upstream handler with the input flow of the downstream handler in order to create the linearly-linked structure of the network services.

The FreeNA system can be divided into the FreeNA client and the FreeNA server. The FreeNA client is used to operate the FreeNA server in user-friendly ways. In practice, the whole FreeNA system acts as an event-driven system based on the operational commands, and the client supports the command input. The FreeNA server performs the core functions like the network service composition and service insertion in accordance with the requests from the client. To ensure the portability of FreeNA, the server divides its tasks into platform-independent and platform-dependent tasks.

To support the system operations, an XML-formatted configuration file is used for each application. After taking into consideration that inexperienced users will be using this system, a functional selection approach was adopted to describe the network services and any additional necessary information. In the configuration file, information on the network services to be inserted, their parameters, the global and local insertion rules, and transport-layer protocols can be written.

Chapter 5

Implementation

The fundamental designs of FreeNA, the system compositions, and their relationships have been explained so far in view of the user experience. In particular, the flow handler (chain) concept was the key idea for composing network services. However, a practical method of linking independent network service libraries and how to interpose the flow handler chain between the application and the operating system have not yet been clarified.

This chapter describes the internal architecture of FreeNA. First, the implementations of the FreeNA client and server are explained, then the mechanism of the service insertion and the implementation of the flow handler are explained. In addition, the mechanism of the Transport-layer protocol insertion is also described as well as that for the upper-layer service insertion.

5.1 FreeNA Client/Server

Since FreeNA aims to work on multiple platforms, most of it is implemented in the Java language. In Figs.4.4 and 4.7, the client and the dotted square on the left of the server (platform independent part) are coded in Java, and the right dotted square (platform dependent part) is implemented as a library coded in C++, because the *interposer* directly manipulates the application process. There are several implementations of the interposer library because the manipulation techniques vary from platform to platform. In either platform, the interposer library is accessed via a Java Native Interface (JNI) using the

Java-coded part.

5.2 Upper-Layer Network Services Insertion

5.2.1 Network Services Composition

As mentioned in the previous section, the network services are implemented as independent shared libraries based on the concept of the flow handler. So, all FreeNA has to do is to bind the input/output data flows of the service libraries, switch from the socket function calls invoked by the application to the library function calls of the uppermost service library, and have the undermost service calling actual socket functions. In this section, a practical method for composing independent network services is explained.

To compose the flow handler chain, the `service_info` C structure shown in Fig.5.1 is defined. Each network service library has one corresponding `service_info` structure instance. The structure contains the service parameters and insertion rule information of the service. Moreover, it contains the function pointers to the functions of the downstream service library and the pointers are set outside of the service library. Therefore, one network service can use another service even though the service library itself does not know the details of another library.

Eventually, FreeNA hierarchically inserts network service libraries between the application and the socket library. Figure 5.2 shows the diagram of the hierarchy. As it can be seen in the figure, not only the service libraries, but also a *Control library* and an *Interface library* are inserted together. The control library dynamically loads all the underlying libraries into the application process and sets up the `service_info` structures using the configuration information passed from the FreeNA server. The `service_info` structures are formed as a linked list and passed down to the downstream library by the `init()` functions. The interface library is inserted to access the intrinsic socket library of the platform. The *SSL library* is also one of the network service libraries. However, it differs from the other libraries in that the SSL library is located as the bottom-most library and used instead of the interface library. We present the implementation details

```
struct service_info
{
    /* Pointer to the downstream service's one */
    struct service_info* next;


    /* Parameter information of this service */
    int    num_of_params;
    char** params;
    char** values;


    /* Service insertion rule contains effective port numbers and
       communication type */
    struct rule tcp;
    struct rule udp;


    /* Function pointers to initialization/finalization functions
       of the downstream service */
    void (*service_init)( struct service_info* );
    void (*service_exit)( void );


    /* Function pointers to corresponding socket-like functions of
       the downstream service */
    socket_t (*service_socket)( int, int, int );
    int (*service_bind)( socket_t, const sockaddr*, socklen_t );
    int (*service_connect)( socket_t, const sockaddr*, socklen_t );
    ...
    int (*service_send)( socket_t, const char*, socklen_t, int );
    int (*service_recv)( socket_t, char*, socklen_t, int );
    ...

};
```

Figure 5.1: The `service_info` structure definition. All member variables are set by a control library.

of the library later.

By comprising FreeNA with the above hierarchical architecture, the arbitrary service libraries can be inserted transparently into the socket function call flow between the

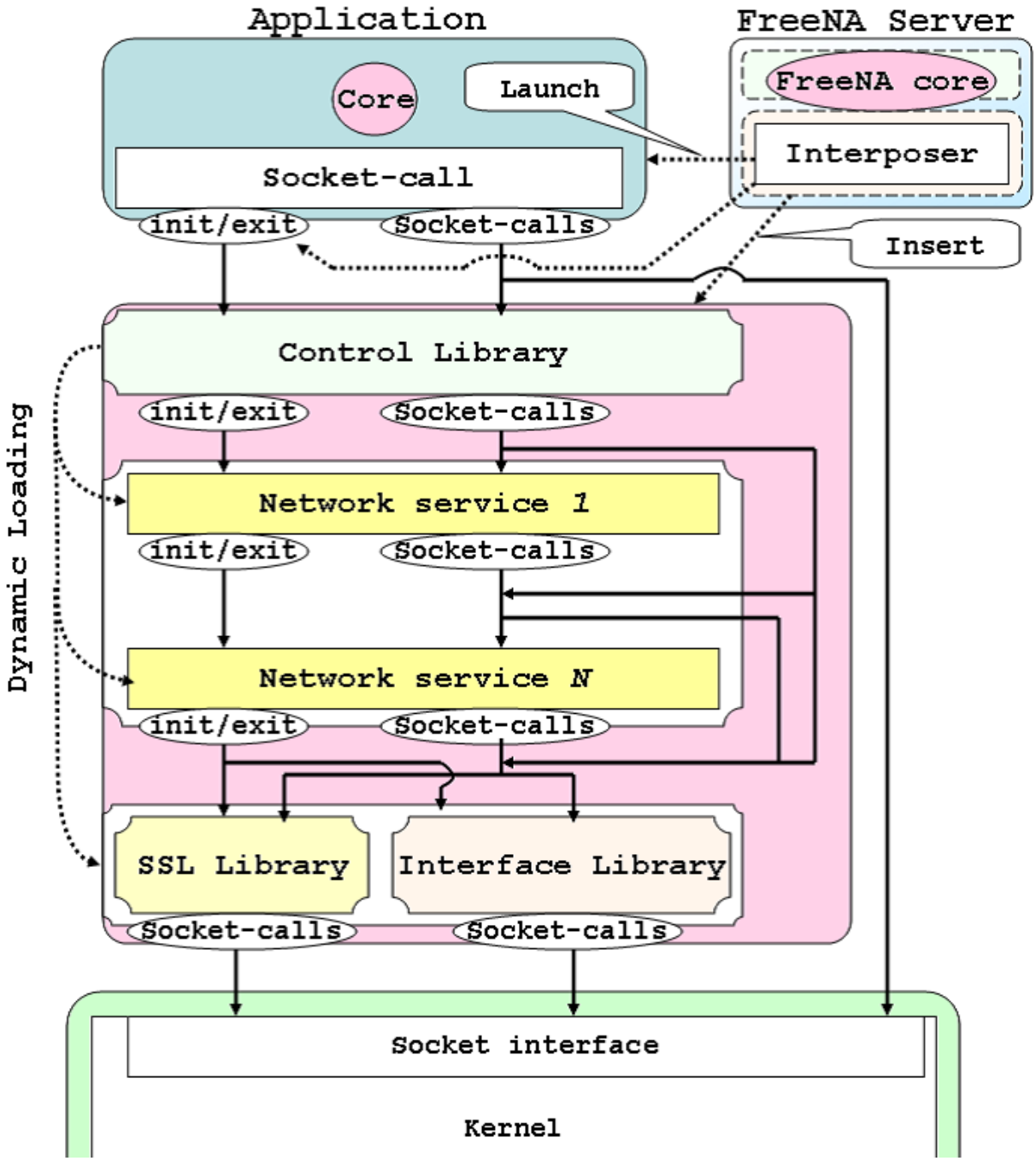

Figure 5.2: Hierarchical Structures of Inserted Services

application and the socket library. Furthermore, FreeNA creates more flexible function call flows.

The service insertion rules mentioned in section 4.4.1 are used to switch the call flows. When the condition is satisfied, the control library calls the service library's function. Otherwise, the control library bypasses the underlying libraries and directly calls the

actual socket functions.

5.2.2 Transparent Insertion Mechanism

So far, the flow of the function call through the control library, service libraries, and interface libraries has been explained. The remaining concern is how to switch from the socket function calls invoked by the application to the control library's function calls.

FreeNA leverages the runtime system-call interposition mechanism to hook the socket functions. This method is more suitable for creating FreeNA's mechanism than Proxy-based interposition and source code level interposition. That is because hooking the socket functions at the process image can ensure there is abetter performance and end-to-end transparency than that of Proxy-based interposition. Moreover, users do not need to have the source code of the application, know the internal structure of the application, nor consider the programming languages like source code level interposition.

In practice, the dyninst API is used for interposition. The API provides a variety of methods for dynamically changing the runtime process image to instrument/remove the CPU instructions into/from the image in an abstract manner.

Figure 5.4 shows a schematic process image of an application using FreeNA. First, FreeNA launches the application and loads the control library into the process image, then a `ctl_init()` function and a `ctl_exit()` function of the library are embedded into the `main()` function of the application before starting. The configuration information is also embedded as the `ctl_init()` arguments. Next, FreeNA switches the function call target from the socket library to the control library by rewriting the process image. The network service libraries and interface libraries are dynamically loaded by the control library at the time of initialization.

Here, since the Interposer is independent of the core FreeNA server, it is possible to call another Interposer library that leverages other interposition mechanisms, such as library preloading and Detours[71]. This alternation is useful when users have platforms that Dyninst API cannot support. Instead, other mechanisms for initialization/finalization and passing configuration information are needed. Therefore, it is better to use Dyninst

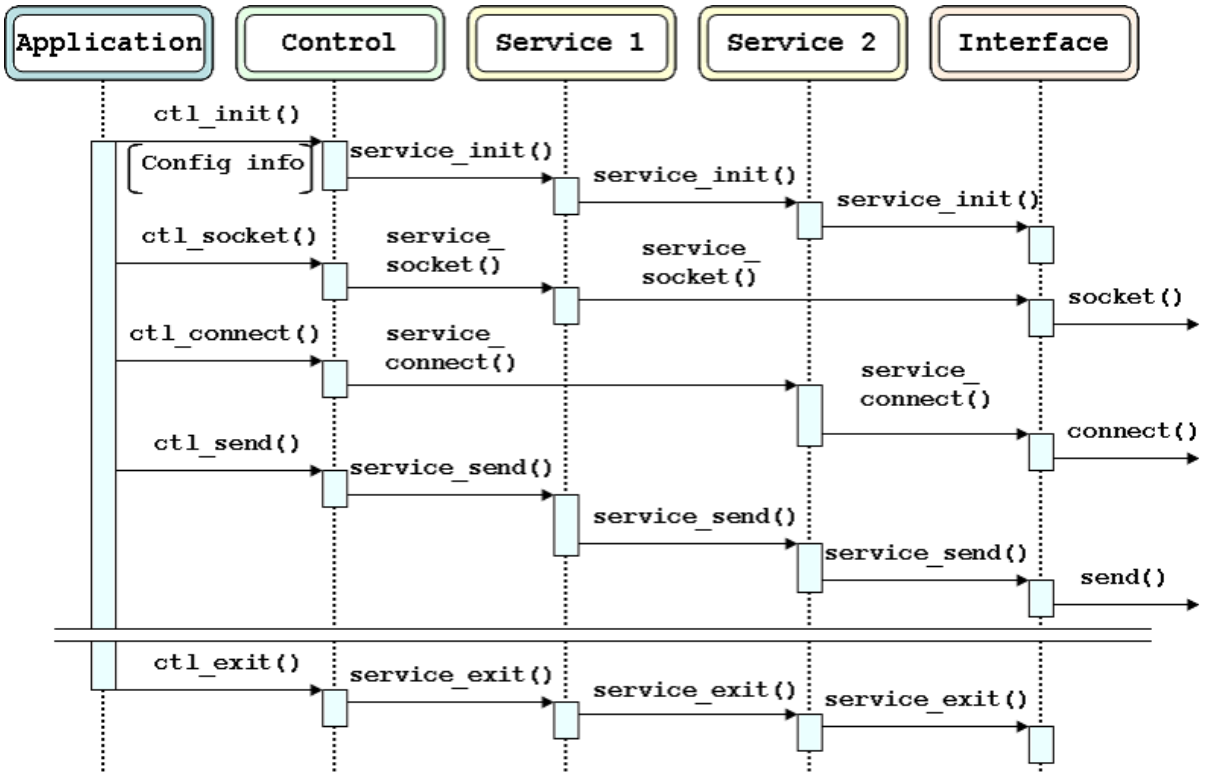

Figure 5.3: Execution sequence of the FreeNA-enabled application

API for the interposition whenever it supports the platform.

5.2.3 Development of Network Services

In this section, the practical code of network services is introduced by looking at some examples of compression service and SSL service. The compression service library is a simple library that processes data during I/O operations. The SSL service library provides a SSL/TLS compliant secure communication mechanism based on the PKI framework to the applications. As we can see later, service library developers can leverage the existing client library, such as OpenSSL[72], libcurl[73], and Zlib[74], to implement the service libraries for FreeNA.

It should be emphasized that the service libraries do not provide APIs for the application developers but instead offer socket-like interfaces. Therefore, the functions of the service libraries have to be implemented to conform to the purpose of the socket functions.

Figure 5.5 shows the condensed code of a compression sending function. As you can see,

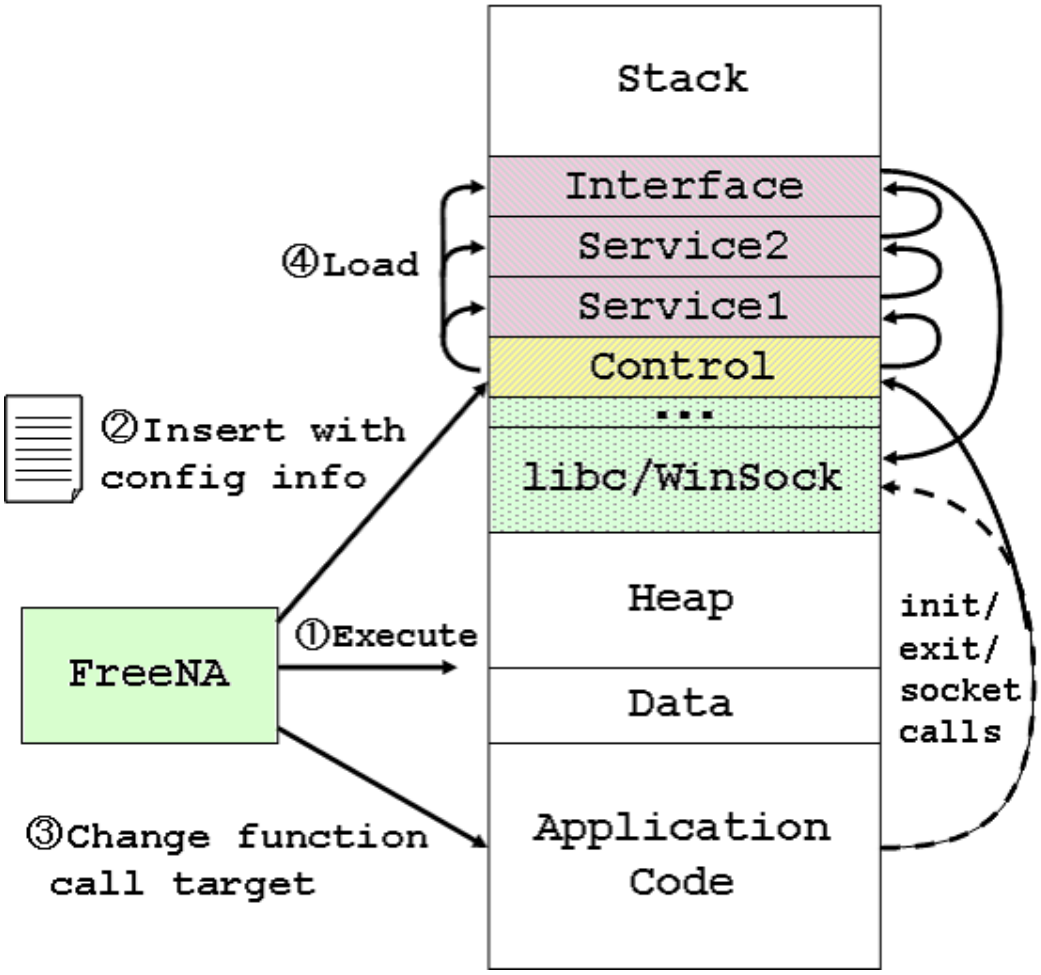

Figure 5.4: Synoptic structure of process image with FreeNA

the `service_send()` function has the same interface as a normal `send()` function. The data passed from the upstream library is compressed and passed down to the downstream library using the `service_info` structure of this library.

Likewise, the code of SSL service library is introduced as a more complicated example. Since SSL is associated with the socket layer, the SSL service library is defined as the lowermost library. Table 5.1 lists the functions of the SSL library and their purposes. `service_init()` is used to initialize a SSL environment just like for loading a certification file, setting random numbers, and determining encryption methods based on the parameter information. `service_connect()` and `service_accept()` establish a SSL connection based on the already connected TCP socket. `service_send()` encrypts the data being sent and passes it down to the socket layer. `service_recv()` gets the received data from

```
ssize_t service_send(socket_t s, const char *data, size_t len, int flags)
{
    ssize_t ret;
    /* Service applied socket is already registered according to the
       local rule */
    if ( use_this_service( s ) ) {
        size_t new_len = BUF_SIZE - HDR_SIZE;

        /* Compress 'data' and output to 'buf' */
        compress( &buf[ HDR_SIZE ], &new_len, data, len );

        /* Set header information (packet length) */
        set_packet_length( buf, new_len );

        /* Pass down 'buf' using 'service_info' structure of this
           library */
        ret = info->service_send( s, buf, new_len + HDR_SIZE, flags );
   }
    else {
        /* Do nothing but pass down 'data' */
        ret = info->service_send( s, data, len, flags );
   }
    return ret;
}
```

Figure 5.5: Example of compression service library's functions

the socket and decrypts the data. `service_close()` disconnects the SSL connection.

Table 5.1: SSL service library's functions and their purposes

Library functions	Purposes
service_init	Initialize a SSL environment
service_connect	Establish a SSL connection as a client based on the connected socket
service_accept	Establish a SSL connection as a server based on the connected socket
service_send	Encrypt data and send them
service_recv	Receive data and decrypt them
service_close	Disconnect the SSL connection

Next, the condensed code example of the SSL service library functions is shown in Fig.

5.6. The SSL service library internally leverages the OpenSSL library[1] and associates the socket with the SSL session object.

```
int service_connect( socket_t s, const struct sockaddr* addr,
                     socklen_t addrlen )
{
    /* Call actual 'connect' socket function */
    int ret = sys_connect( s, addr, addrlen );

    /* Setup a SSL object */
    SSL_CTX *ctx = setup_client_ctx();
    BIO* bio = BIO_new_socket( s, BIO_NOCLOSE );
    SSL* ssl = SSL_new( ctx );
    SSL_set_bio( ssl, bio, bio );

    /* Make a SSL connection */
    SSL_connect( ssl );

    certification_check( ssl, addr );

    /* Associate SSL object with socket */
    register_socket( s, ssl );

    return ret;
}

ssize_t service_send( socket_t s, const char* data,
                      size_t len, int flag )
{
    /* Get the SSL object associated with the socket */
    SSL* ssl = get_SSL( s );

    /* Encrypt data and send them */
    ssize_t n = SSL_write( ssl, data, len );

    return n;
}
```

Figure 5.6: Example of SSL service library's functions

[1]Customized OpenSSL library is used so that the library calls our prepared function instead of the original socket functions to prevent recursive socket functions from calling (See B.1)

5.2.4 Handling Multiple flows

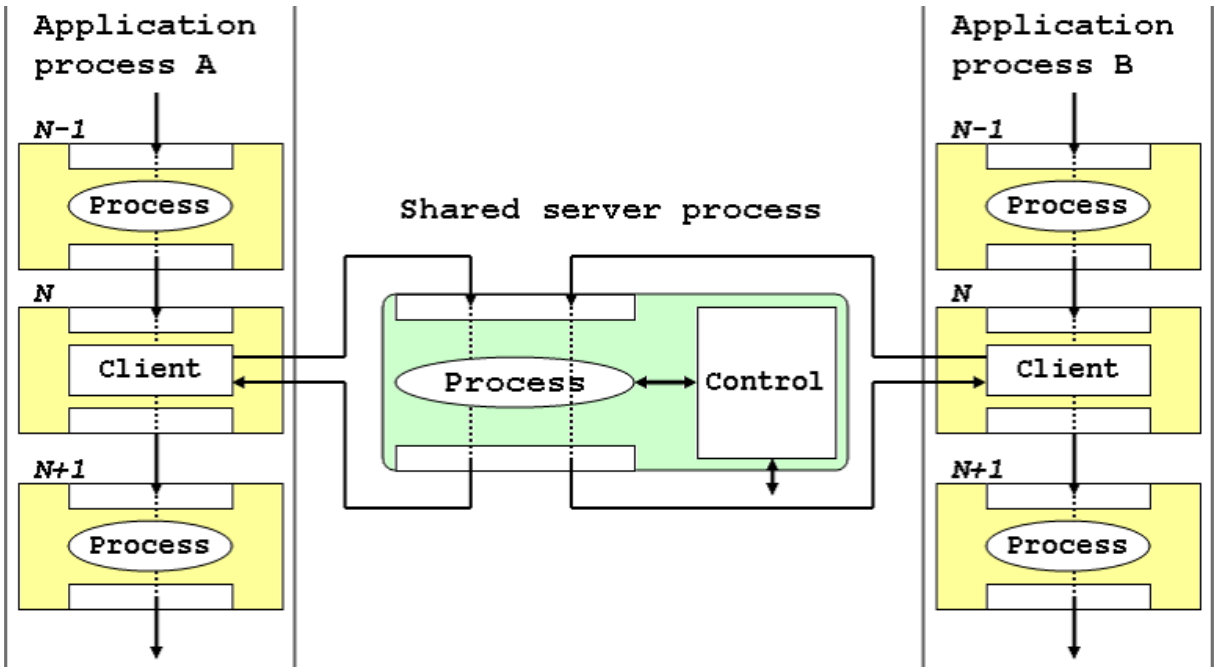

Figure 5.7: Internal structure of the shared server and its service library

Some services should be applied to multiple flows across the processes as a group; for example, a congestion control service may use a shared network status in order to limit the transmission rates of the communication flows that have the same network path. While many related systems handle communication flows separately, TESLA[23] supports the mechanism for handling flows across multiple processes. Network services for TESLA are separated as independent processes from the application processes, and these processes are linked by the internal socket communication. However, since this separation structure is applied to the services that handle single flows, significant performance overhead is induced.

Therefore, FreeNA supports a combination of the application-specific (local) services and application-independent (global) services. The local service is a normal service as described before that is applied to the communication flows of a single application. While, the global service is implemented as an independent process appropriately named a shared server and handles the communication flows across multiple processes like TESLA's ser-

vices. Figure 5.7 shows the structure of the shared server and the relationship to the application processes. In the figure, service *N* is the global service, and services *N-1* and *N+1* are the local services. When the function of service *N* is called, this execution flow is moved to the corresponding shared server using the function parameters through the internal socket communication. After the service processing, the resultant data is backed to the service *N*, and consequently passes down to service *N+1*. Note that the shared server should work under the principle of a socket interface.

5.3 The Negotiation Mechanism

The negotiation mechanism is introduced to FreeNA in order to dynamically compose network services. In this section, when FreeNA conducts the negotiation, what communication channel is used for the negotiation and how to decide the network services by the negotiation are described.

5.3.1 The Timing of the Negotiation

Generally, network applications call `connect()`/`accept()` socket-calls to establish a connection before starting a communication. When considering that the establishment of a connection is the first contact process to the remote application, the negotiation by an end-to-end FreeNA should be conduced at that time. Since FreeNA has the interception mechanism for arbitrary socket-calls, it can conduct the negotiation while interrupting the `connect()` or *accept()* socket-call.

Figure 5.8 shows the sequential flow for conducting the negotiation. In processes (1) and (2), FreeNA interrupts the execution of the socket-calls invoked by the both applications. In process (3), FreeNA conducts the negotiation to decide on the network services. After the negotiation, the interrupted socket-calls are executed to establish the TCP connection. Here, one question arises, why the negotiation is conducted before the connection establishment process. The reason for this is that the communication channel for the negotiation can differ from the channel for the application-level communication

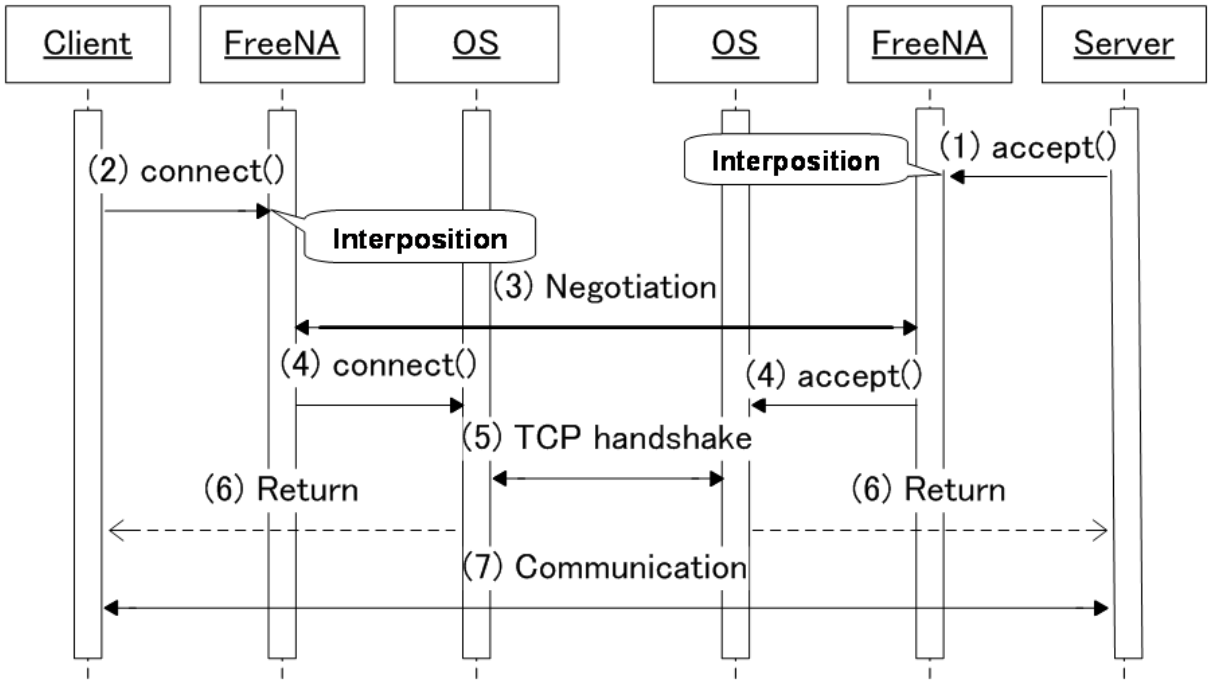

Figure 5.8: Timing of conducting the negotiation

as described later[1]. In the process (6), the result of the connection establishment is returned to the application as a return value of the socket-call. Finally, the end-to-end applications communicate with each other using the same network services. Note that if the applications use the UDP protocol, the negotiation should be conducted while interrupting the firstly-invoked `sendto()`/`recvfrom()` socket-calls.

5.3.2 Negotiation Channels

Since the negotiation is conducted before the connection establishment for the application-level communication, a dedicated communication channel for the negotiation is needed. This separation of channels is effective when taking into consideration the non-FreeNA-enabled applications. That is, if one end of FreeNA tries to negotiate with the non-FreeNA-enabled application through the application-level communication channel, a normal communication cannot be expected. In this study, the three types of negotiation channels mentioned below are assumed.

[1]It is possible to share the TCP connection for the negotiation and the communication.

Outer Channels with SIP Protocol

The Session Initiation Protocol (SIP)[75] is widely used to establish the session between end-to-end systems, and provides a user management function, a location function, a session management function, and an authentication function. One advantage of SIP is that it can seamlessly cooperate with other protocols and services. For example, many multimedia network applications like video streaming often use a Session Description Protocol (SDP)[76] with an SIP protocol to confirm the media attributes.

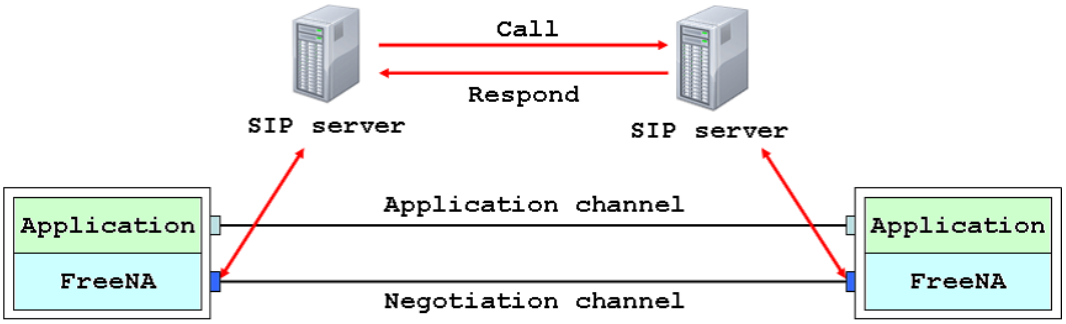

Figure 5.9: Dual channels for the negotiation and the communication with SIP protocol

FreeNA leverages the flexibility of the SIP protocol to establish the negotiation channel depicted in Fig. 5.9. Since the SIP protocol is already standardized, its session establishment mechanism is available on existing networks. Therefore, only the negotiation internals can be considered in this study.

Outer Channels under Small Network

When FreeNA is deployed in small autonomous network environments, the SIP protocol may be a too full-fledged mechanism for session establishment. Consequently, it is better to establish a negotiation channel by directly specifying another port number without the SIP protocol.

Inner Channels

The two previously mentioned methods require additional establishment processes and port numbers for each application. This method enables FreeNA to negotiate with

a remote FreeNA using the application-level communication channel while maintaining a compatibility with the non-FreeNA-enabled applications. To share the channel for both the negotiation and the communication, the received message must be transparently distinguished between the negotiation message and the application message. However, straightforward tunneling techniques that embed the negotiation protocol into the application protocol cannot ensure the interoperability.

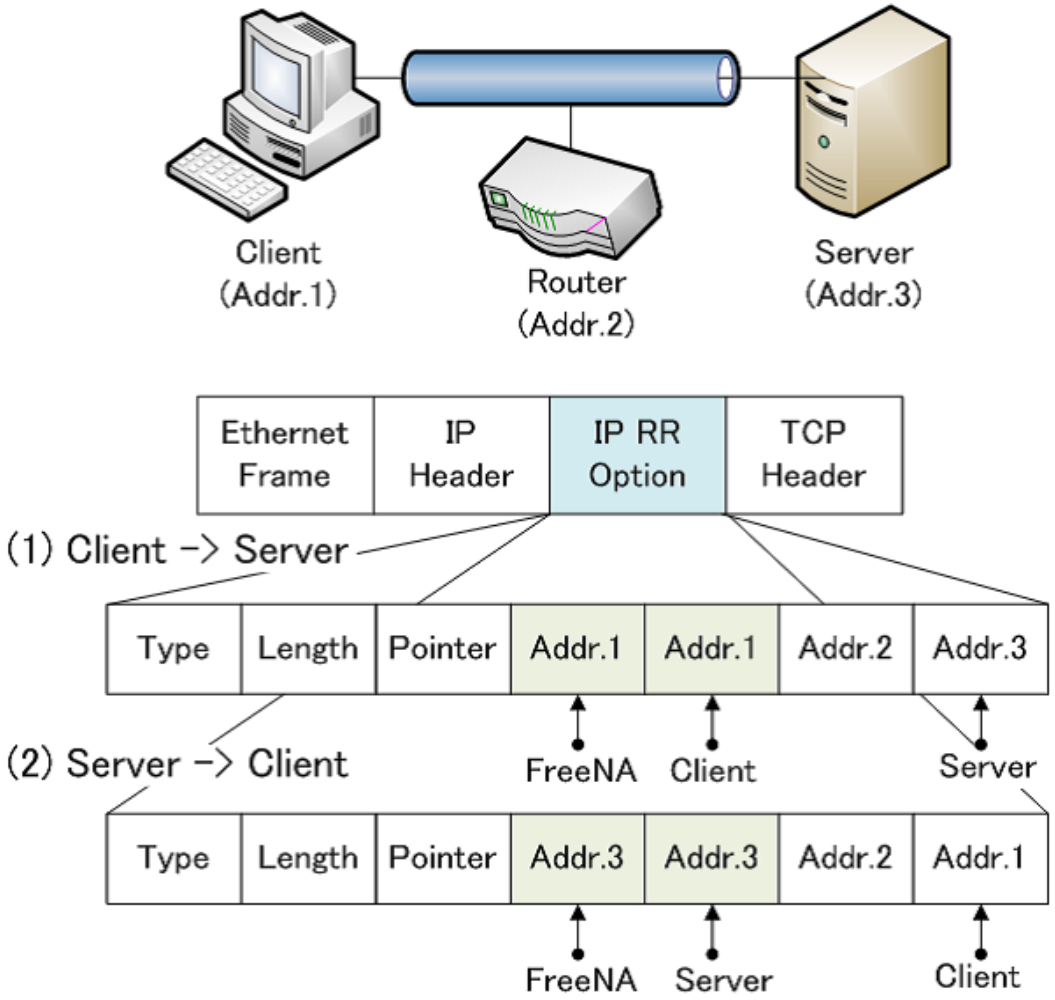

Figure 5.10: Determination of FreeNA existence with IP RR

In this study, a record route mechanism of the IP protocol[77] is applied to ensure the interoperability. The IP record route option sequentially records the IP addresses of every node between the end-to-end systems. Then, the server-side of FreeNA can recognize the existence of the FreeNA on the client-side by appending the IP address of the FreeNA

during the time of the establishment of the connection. As illustrated in Fig. 5.10, FreeNA is regarded as a node between the communication paths, and it records the IP address of the underlying host to the head of the record route list. That is, the same IP address is recorded twice in the list (The first address represents FreeNA and the second address represents the sender-side node.), and the receiver-side FreeNA compares the first two addresses in the list. If the two addresses are the same, FreeNA conducts the negotiation.

5.3.3 Determining Network Services

The concern that remains for the negotiation is how to systematically determine appropriate network services. First, a descriptive format for the negotiation message that will contain the network service information is required. So, the SDP protocol is utilized to systematically describe the service information. Since the SDP protocol allows for the extension of one's own format, the dedicated format for the network service information is defined as that shown in Fig. 5.11.

```
v=<version number>
o=<user name> <session id> <version> <nettype> <addrtype> <unicast addr>
s=<session name>
c=<nettype> <addrtype> <connection addr>
t=<start> <stop>
m=application <port number> SERVICE/<service name> <essentiality>
a=fmtp:<parameter name> <parameter value>
```

Figure 5.11: Extended SDP for FreeNA's negotiation

While the meanings of the **v**, **o**, **s**, **c**, and **t** tags are not changed, another interpretation of the **m** tag and the **a** tag is introduced to represent the network service information. In practice, the name of the service, the local port number of the channel, and an essentiality attribute are specified in the line of the **m** tag. The essentiality attribute represents the importance of the services during the communication, and corresponds to the service type and parameter block of the configuration file. Table 5.2 lists the relationship between the essentiality attribute and the configuration type. In the line the of **a** tag, a parameter

Table 5.2: The relationship between the essentiality attribute and the configuration type

Essentiality	**Service type**	**Description**
required	**global(/required)**	Specified service must be used at the communication
selectively-required	**global(/required)**	In addition to **required**, one of parameter value must be selected from the parameter blocks
optional	**global/optional)**	Specified service can be omitted by the negotiation result
selectively-optional	**global/optional)**	In addition to **optional**, one of parameter value must be selected from the parameter blocks

name and value pair of the service is specified with the **fmtp**(format-specific parameters) attribute of the SDP protocol.

The SDP protocol does not provide a negotiation mechanism for the session originally, but Rosenberg et al.[78] have proposed an offer/answer model using SDP to negotiate the session attributes. Therefore, FreeNA's negotiation is conducted in the proposed offer/answer model. The basic pattern of the negotiation is as follows. First, the client-side (Offerer) shows the supportable service compositions to the server-side (Answerer) using the extended SDP protocol. The SDP message is automatically composed from the client-side configuration file. Next, the answerer compares the offered service composition with the available composition by the answerer. If all the **required** services are available to the answerer, the negotiation ends in success with the answerer responding with its SDP message that contains the network services to be used during the communication. If the negotiation failed, FreeNA does not insert any service into the application.

Figures 5.12 and 5.13 show examples of the negotiations in the offer/answer model. The offer message states that the offerer wants to use a cryptography service and a compression service during the communication, but the answerer must select a combination of the parameters for the cryptography service because the **selectively-required** essentiality is specified. Moreover, the compression service can be omitted during the communication because the **optional** essentiality is specified.

Then, the answer message states that the cryptography service is only used during the communication with a 256-bit AES algorithm in the CBC mode. The answerer denies the other parameter combination and the compression service by specifying port number 0. Finally, this negotiation ended in success because all the **required** services are available by both the offerer and the answerer.

```
v=0
o=clt 844526 844526 IN IP4 clt.example.com
s=-
c=IN IP4 clt.example.com
t=0 0
m=application 10000 SERVICE/CRYPTO  selectively-required
    a=fmtp:algorithm AES
    a=fmtp:key_size 256
    a=fmtp:key_size 128
    a=fmtp:mode CBC
m=application 10000 SERVICE/CRYPTO  selectively-required
    a=fmtp:algorithm DES
    a=fmtp:key_size 128
    a=fmtp:mode CBC
m=application 10000 SERVICE/COMPRESSION  optional
a=fmtp:algorithm Deflate
```

Figure 5.12: Offer message with extended SDP

```
v=0
o=svr 844564 844564 IN IP4 svr.example.com
s=-
c=IN IP4 svr.example.com
t=0 0
m=application 8000 SERVICE/CRYPTO selectively-required
    a=fmtp:algorithm AES
    a=fmtp:key_size 256
    a=fmtp:mode CBC
m=application  0 SERVICE/CRYPTO selectively-required
a=fmtp:algorithm DES
a=fmtp:key_size 128
a=fmtp:mode CBC
m=application  0 SERVICE/COMPRESSION optional
a=fmtp:algorithm Deflate
```

Figure 5.13: Answer message with extended SDP

5.4 Transport-layer Protocol Insertion

5.4.1 Composition of Transport-layer Protocol

From this section, the details of the insertion mechanism of the transport-layer protocols and how to achieve a transport-layer protocol-free environment are explained.

```
int main()
{
    ...
    socket_t svr_sock = socket( ..., IPPROTO_X );
    bind( svr_sock, ... );
    listen( svr_sock, ... );
    socket_t clt_sock = accept( svr_sock, ... );
    ...
    send( clt_sock, ... );
    ...
}
```

Figure 5.14: New protocol is added within the kernel, and users can specify it by a socket parameter

Generally, there are two methods for implementing alternative transport-layer protocols in traditional TCP/UDP protocols. One is where the protocols are directly implemented within the kernel, and the users can use them by identifying the proper socket parameter within the application source codes, as shown in Fig.5.14.

```
int main()
{
    ...
    socket_t svr_sock = protocol_socket( ... );
    protocol_bind( svr_sock, ... );
    protocol_listen( svr_sock, ... );
    socket_t clt_sock = protocol_accept( svr_sock, ... );
    ...
    protocol_send( clt_sock, ... );
    ...
}
```

Figure 5.15: New protocol is added within the userland as a library, and users can use it by calling library functions

The other method is where a raw IP mechanism[79] is used to implement the protocol in the user-space. Raw sockets are different from normal STREAM/DGRAM sockets, because they allow developers to directly create raw IP packets. Generally, only privileged users can use raw sockets and developers have to manage the port numbers on their own.

Figure 5.16: New protocol is added within the userland as a library, and users can use it by just calling traditional socket-calls as they are

On the other hand, FreeNA leverages the raw IP mechanism to implement the additional transport-layer protocols. FreeNA is different from similar systems in that the protocol functions are implemented as independent programs and accessed by the applications transparently using the socket-call interposition mechanism.

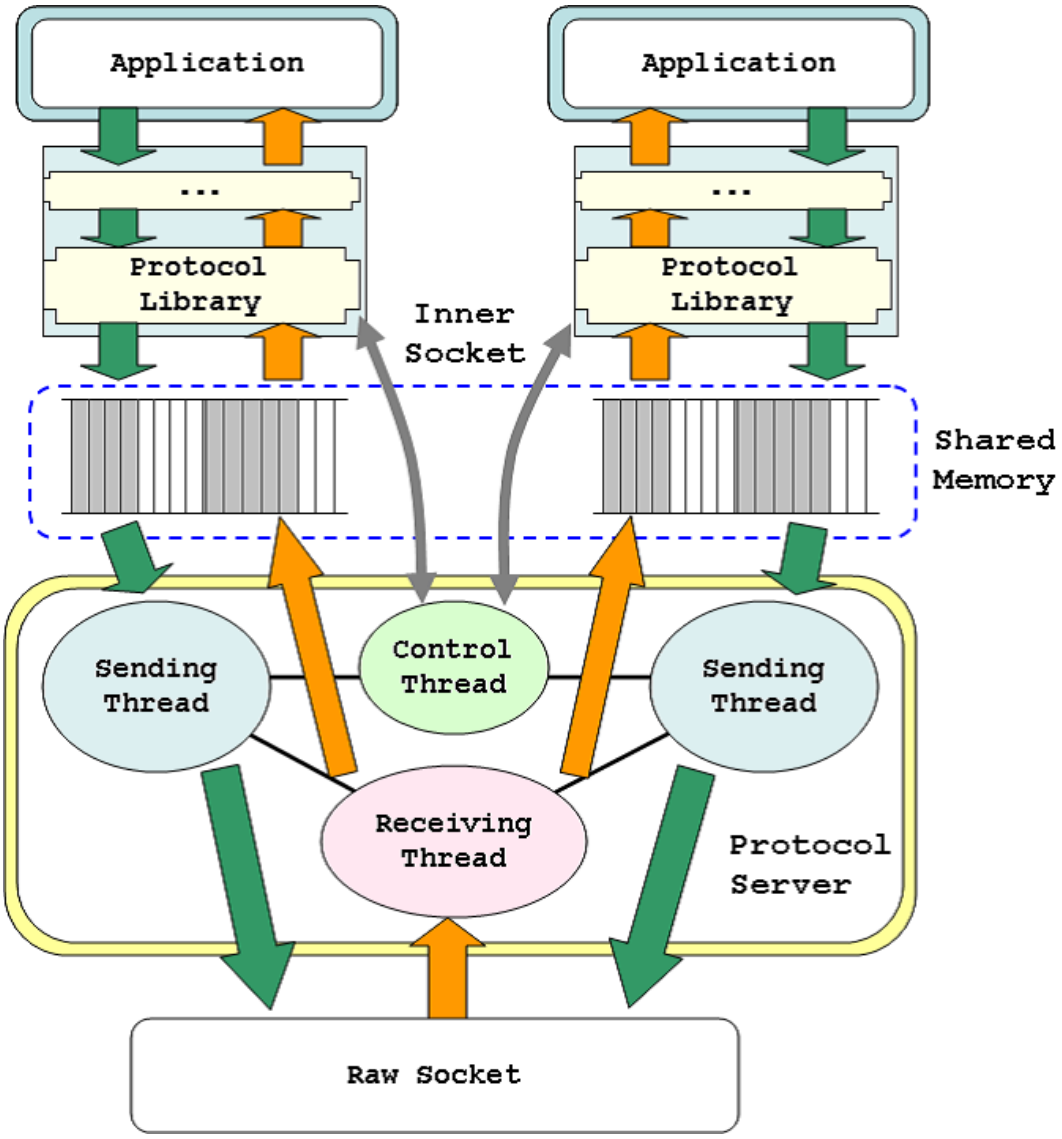

Figure 5.17: An implementation structure of transport-layer protocols

5.4.2 Development of Transport-layer Protocols in User-space

Figure 5.17 shows the implementation structure. A protocol server and a protocol library are implemented in order to construct the transport-layer protocols in the user-space. The former is an independent program and executes the core processes of the protocol, such as packet transmission/receiving, connection management, and port management. An independent protocol server is needed because the raw sockets do not have the concept of port numbering or multiplexing. The latter bridges the protocol server and the application process. The interface of the protocol library is shown in Table 5.3.

Table 5.3: Relationship between the protocol library and server process

Library function	Comm. method	Server's process
protocol_socket	Inner socket	Register socket info.
protocol_close	Inner socket	Remove socket info.
protocol_bind	Inner socket	Specify address info
protocol_connect	Inner socket	Establish a connection
protocol_listen	Inner socket	Listen a port
protocol_accept	Inner socket	Accept a client
protocol_send	Shared memory	–
protocol_recv	Shared memory	–
protocol_sendto	Shared memory	–
protocol_recvfrom	Shared memory	–

The protocol library provides several functions (**protocol_***) that have the same interface as the socket functions. When the application calls a socket function, it is hooked by FreeNA and the corresponding library function is executed instead. The control information and data packet are passed to the protocol server via the inner socket or shared memory.

5.4.3 Process Flow from the Application to the Protocol Server

In this section, the practical execution steps of the socket-calls through the application to the protocol server are explained. Since FreeNA inserts several libraries into the application process, the relationship between these libraries should be clarified.

Figure 5.18 extends Fig.5.2, and shows the hierarchical structure of the application process when additional transport-layer protocols are inserted. Unlike the previous structure, the interface library switches the actual socket functions and inserted protocol library's functions. When a protocol within the kernel is used, the interface library directly calls the socket functions just as before. Otherwise, the interface library calls the functions of the protocol library to access the corresponding protocol server.

5.4.4 Protocol-free Environment

Generally, the socket interface provides two types of functions for the connection-oriented protocol and the connection-less protocol. Since the socket interface is designed

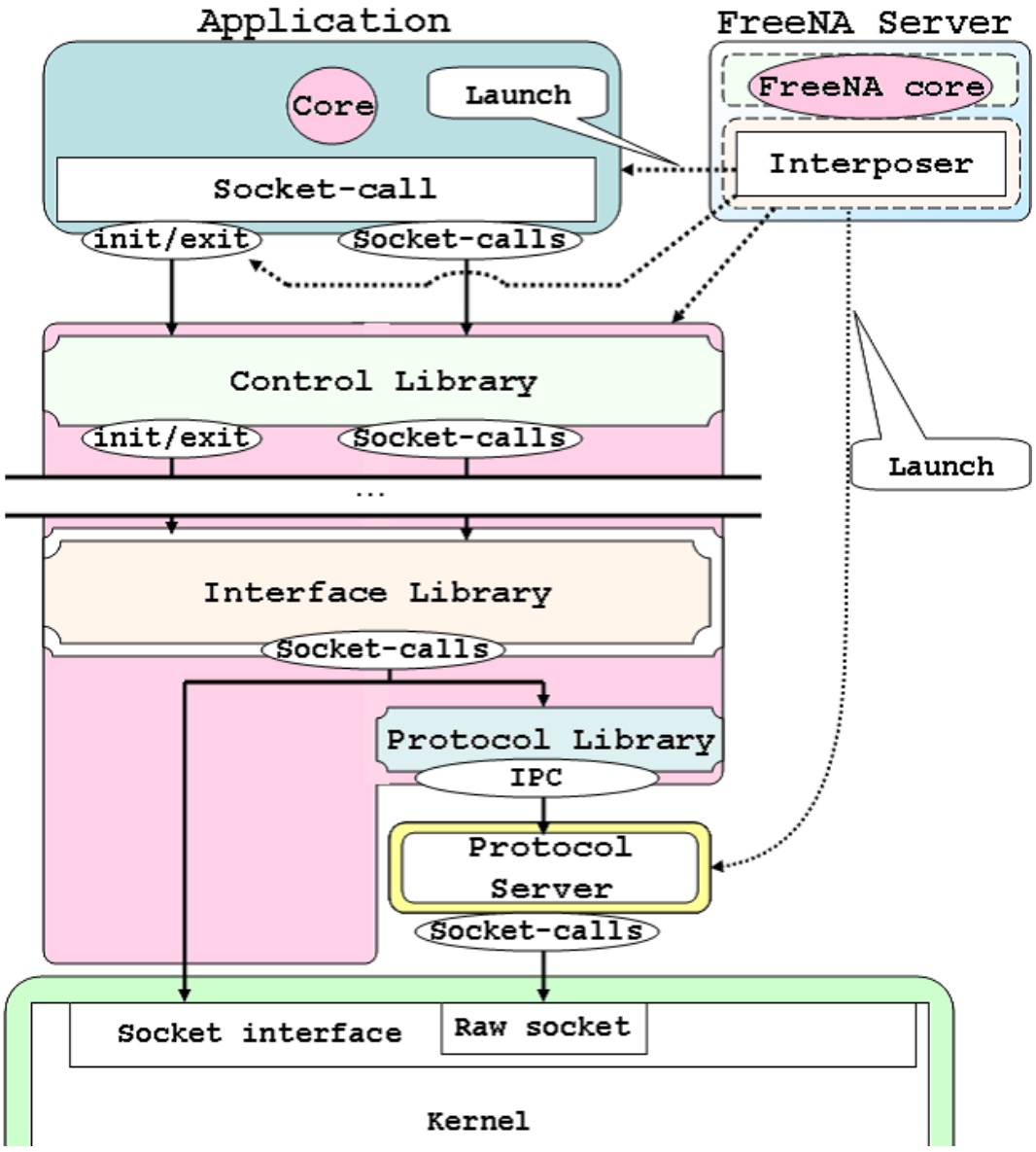

Figure 5.18: An insertion structure of service functions and transport-layer protocols

without any dependency on a certain protocol, the functions of additional protocols other than TCP/UDP can also be used via the socket interface conceptually. In this study, the adaptive mechanism for switching the transport-layer protocol used by server-type applications to another protocol used by the client-type application is implemented by leveraging the abstract design of the socket interface.

Actual execution steps of the protocol switching are explained as follows by taking a code example of the `service_accept` function within the interface library shown in

```
socket_t service_accept( socket_t tcp_sock, ... )
{
    /* Create and setup a socket for SCTP */
    socket_t sctp_sock = protocol_socket( ... );
    protocol_bind( sctp_sock, ... );
    prootcol_listen( sctp_sock, ... );

    fd_set rfds;
    FD_ZERO( &rfds );
    FD_SET( tcp_sock, &rfds );
    FD_SET( sctp_sock, &rfds );

    select( max( tcp_sock, sctp_sock ) + 1, &rfds, ... );

    if (FD_ISSET(tcp_sock, &rfds)) {
      /* The client connected to the server with TCP */
      clt_sock = accept( tcp_sock, ... );
      /* TCP is activated */
    }
    else if ( FD_ISSET( sctp_sock, &rfds ) ) {
      /* The client connected to the server with SCTP */
      clt_sock = protocol_accept( sctp_sock, ... );
      /* SCTP is activated */
    }
    return clt_sock;
}
```

Figure 5.19: Checking a protocol used by the client at connection time

Fig.5.19 into account.

- The server-type application uses the TCP protocol as a default, and the SCTP protocol is also available as an alternative protocol.
- The application executes the `socket`, `bind`, and `listen` socket-calls, and a TCP socket is in the waiting state.
- The application calls the `accept` socket-call, and the `service_accept` function within the interface library is internally invoked.

- Another socket for the SCTP protocol is created by executing the appropriate functions of the SCTP protocol library.
- Within the `service_accept` function, a `select` system call is called to observe both the TCP socket and the SCTP socket.
- If the client accesses the server using the TCP protocol, a `select` system call returns, and the TCP socket is set as the connection message has been arrived.
- If the client accesses the server using the SCTP protocol, the `select` system calls returns, and the SCTP socket is set(The SCTP protocol server sends a signal to the protocol library via the internal socket channel when the SCTP connection is established).
- After connection establishment, the application can use the same protocol as the client's one by calling the socket-calls as usual.

In this way, the transport-layer protocol-free environment for the connection-oriented protocol can be established. In addition, this mechanism is also applicable to connectionless protocols. Although these protocols do not require `accept` socket-call, a `recvfrom` socket-call can be the alternative such that the `select` system-call is called in the first round of the `recvfrom`.

5.5 Summary

In this chapter, the practical mechanism of the FreeNA system including the client/server systems, the flow handler (chain) concept, the transport-layer protocol insertion, and the protocol-free environment were described.

The entire client and the largess of the server are implemented in Java to ensure the system portability. The interposer components are implemented in C++, and are prepared for each platform to directly manipulate the application process. The interface between the server and the interposer is implemented using JNI mechanism.

To compose the flow handler chain structure, each network service library must have the `service_info` structure instance to access the downstream library. FreeNA inserts not only the network service libraries, but also the control library and interface library between the application and the socket interface hierarchically. The control library dynamically initializes all the underlying libraries at startup. The interface library is used to call the intrinsic socket functions or protocol library functions.

Internally, the dyninst API is used mainly for the system-call interposition. The API provides a variety of methods for dynamically changing the runtime process image in an abstract manner. It is also possible to use another interposition mechanism like library-preloading by implementing the mechanism to another interposer component.

FreeNA also allows users to insert transport-layer protocols by leveraging a raw IP mechanism. The protocol library and independent protocol server programs are implemented in order to construct the transport-layer protocols into the userland. The protocol server executes the core functions of the protocol, and the library works as the interface between the application process and the server process.

Moreover, the adaptive mechanism of switching the transport-layer protocol used by the server-type applications to another protocol used by the client-type application is implemented by leveraging the abstract design of the socket interface. In practice, `select` system-call is used to decide on the appropriate protocol for the current communication at the time of access.

Chapter 6

Evaluation

In this chapter, the FreeNA system is evaluated from diverse perspectives to show the fulfillment of the FreeNA's goals. The functionality of FreeNA is compared with other similar systems. Considering FreeNA is implemented as a comprehensive system, the manageability and security are also discussed. Furthermore, the system performance of FreeNA is measured. Naturally, the application-level performance can be decreased compared with the original application because the additional computation processes for the network services are introduced. Therefore, the performance overhead of the service inserting mechanism itself is evaluated.

6.1 Functionality Comparison

The functionality of FreeNA is an important result in this study in order to achieve a flexible, user-oriented, and adaptive communication framework. FreeNA's functionality can be viewed from various perspectives, such as the users' perspective, the applications' perspective, and the platforms' perspective.In this section, each aspect of FreeNA's functionality is reviewed by comparing it with similar systems introduced in Chapter 2.

6.1.1 Reviewing from Users' Perspective

As described before, FreeNA is designed for both professional and inexperienced users. Therefore, the usability of FreeNA is evaluated by focusing on the configuration mechanism and system operability.

The configuration mechanism is introduced to FreeNA in order to provide a flexible composition method for the application capabilities, to prevent users from being required to have technical and programmatic knowledge. In practice, the XML-formatted configuration file is offered to specify the network services to be inserted, their composition, and parameters. It should be noted that the configuration file can be composed by only selecting the fragmentary functional name. Therefore, users do not need to understand the internal service composition or insertion mechanisms.

FreeNA also provides a user-friendly system operational method as a dedicated client program to hide the complexity of FreeNA's processing. One of the advantages of this client system is that system operability and functionality can be adjusted based on the user's experiences, such that a command-based systems provide finner operations and a browser-based system provides simple and straightforward operations.

The network services of MetaSockets are implemented as the `Filter` thread classes, and they are composed as a pipeline structure in the meta-level socket classes. Users generally prepare the **Decision Maker** classes that monitor the environment and dynamically arrange the inserted filters based on a simple set of rules. Unlike FreeNA, these rules have to be reflected to dedicated DM class implementations such that the DM invokes conditional methods to insert the specified filter. Therefore, programmatic approaches are required to fundamentally modify the application's behavior.

TESLA provides a wrapper program (`tesla`) that sets up the environment for the users. Basically, `tesla` has a handler (network service) configuration, which is an ordered list of handlers, and internally composes the flow-handler chain structure. Users can specify the network services and parameters as specifying arguments to the command-line programs. Having said that, `tesla` only provides on-the-fly usage, rather than providing finer and persistent system management methods.

DITOOLS can be driven by a configuration file by taking into consideration its interposition mechanism. Since an additional service is inserted with the extension of the linker/loader, users have to specify which service library is inserted and where the con-

figuration file is in the runtime system. In the configuration file, the service library name and insertion mode (wrapper mode or callback mode) are specified with the command-based notation. The file itself is specified by an environment variable. Therefore, users are required to know the extension mechanism, and to manage the environment variable properly for multiple DITOOLS-enabled applications.

Users can insert interposition agents into the applications by redirecting the specified system-calls on the BSD/Mach platforms. A C++ toolkit is offered to agent developers to present the primary system interface abstractions (pathnames, descriptors, and files) as C++ objects. Agents can change these abstractions by using the class inheritance. At runtime, an agent loader program loads the user application with the specified agent using the `task_set_emulation()` and `execve()` system calls.

VTL provides a developer toolset for packet capturing, packet manipulation, packet creation, and protocol state maintaining. When taking into account that these mechanisms are implemented as API, VTL does not offer a user-level configuration or system operation mechanisms.

Alpine was designed to allow for network protocol development at the user-level. Therefore, a user-friendly configuration mechanism or the interface tools are not provided.

Trickle offers an application-level traffic shaping mechanism to applications by using the library preloading technique. Users can use the command line utility program to set-up the runtime environment for trickle-enabled applications, and some parameters are also passed via the command line options.

DR-TCP dynamically upgrades the TCP protocol to a newer version depending on the network environment. If the appropriate protocol is not installed on the host, it automatically downloads and installs the protocol from the web. Therefore, users do not need to operate the system.

Table 6.1 outlines a comparison of the system usabilities. The **user-level configuration** item evaluates how easily users can configure the network service composition

without depending on the direct programmatic configuration. The **user-level tools** item evaluates whether the system provides the utility tools for the application invocation and service insertion. Without such tools, the users are solely responsible for these processes, which restricts the potential system users to only experienced developers.

In the table, the '√' mark represents full support, an 'L' mark means limited support, and the '–' mark expresses no support.

Table 6.1: Comparison of system usability

General-purpose systems				
System	**User-level configuration**		**User-level tools**	
FreeNA	√	• XML-formatted file • Functional selection • Detailed configuration	√	• Client program • Operational commands
Meta-Sockets	L	• Adaptation rules • Programmatic configuration • Detailed configuration	L	• Decision Maker class • Adaptive Java
TESLA	L	• Functional selection • Service and parameters	√	• Wrapper program
DIT-OOLS	L	• Command-based files • Service and insertion modes • Environment variables	√	• DI runtime program
Interposition agents	–	• Programmatic configuration • Class inheritance	√	• Agent loader
VTL	–	• Programmatic configuration	–[1]	• Network Interface API • Packet Access API • Protocol State API
Single-purpose systems				
Alpine	–	• Programmatic configuration	–	
Trickle	√	• Command-line options	√	• Utility program
DR-TCP	–	• Programmatic configuration	√	• Autonomous

[1] Users can also use services implemented as executable processes only for VNET[80] extension.

6.1.2 Reviewing from Network Services' Perspective

Network services are the essential components of this study, and therefore, their diversity and composability have significant influence on the usefulness of the entire system.

Each network service in FreeNA is provided as an independent program library in view of the separation of concerns, flexible composition, and service deployment. The separation of concerns is achieved by structurally dividing the application core functions and service functions.

The application itself is never destructed when introducing extended functions because the library content is incorporated in the application's process image not the binary executable file or the source code.

Given the general-purposed service framework, a variety of network services can be supported, and in particular, multiple services can be composed concertedly at any one time even though they are independent of one another. FreeNA supports a range from transport-layer services to application-specific services, and the flow handler concept allows these services to be flexibly combined.

Service developers can implement the library according to the predetermined library interface, such as the socket interface and `service_info` structure of FreeNA. Since each network service does not rely on any other library as a binary program, it is possible for users to acquire the necessary services from third-party service developers.

The configurability of each network service is also an important aspect. For instance, a cryptography service should support various combinations of encryption algorithms, key lengths, and cipher modes, and the users can specify any one of these combinations. FreeNA enables service libraries to have arbitrary `"name=value"` formatted parameters. The semantics of these parameters are interpreted in the service library, and FreeNA just passes the parameter strings from the configuration file to the library as the arguments of the `service_init()` function.

MetaSockets also enables the separation of the network services (filters) from the application, and the inserted filters are combined as a pipeline. Unlike FreeNA, each

filter is implemented as an Java object instantiated `Filter` class. The independence of the filters is achieved by inter-threading the communication via the shared buffer between filters. Each filter can be configured by defining the parameter setting methods. Although complicated parameters can be set, users have to arrange these methods so that `Decision Maker` properly invokes them.

TESLA originally takes advantage of the flow handler concept to transparently compose session-layer network services. However, the implementation form of the flow handler is different in that the service for TESLA is implemented as an instance of the `flow_handler` C++ class, and a couple of handlers are combined in a master process by the service type. The master processes are chained by the inter-process communication because some services require multiple flows across the processes. When taking into account the service deployment, users cannot utilize arbitrary services by just downloading them from third-parties because each handler C++ object must be statically linked to the master program in advance.

Extension functions for DITOOLS are implemented as shared libraries like FreeNA, and the library interface can be assumed to be a socket interface if the network function is extended. Therefore, a variety of network services can be easily deployed to users. Although, DITOOLS supports recursive service insertion by DI runtime, the glue interface between inserted services is not provided. In addition, the parameter setting is not supported.

Since interposition agents are based on the system-call redirection mechanism, agent programs have to be loaded within the address space of the application. Therefore, agents should be implemented as libraries that provide system-call-like functions. Interposition agents have some restrictions in that they cannot be stacked hierarchy because `htg_unix_syscall()` directly invokes kernel functions. That is, only single agents can be loaded to the application at a time.

The network services for VTL are based on packet manipulation and packet injection mechanisms. Therefore, VTL can support data-link layer services to application specific

services. A network service is implemented as an executable program for a half duplex service and two executables for full duplex. However, VTL supposes a single service composition by design.

Alpine is dedicated to the user-level transport protocol service, and it cannot stack upper-layer services on the transport protocol. Unlike FreeNA's method, user-level transport protocols are implemented as shared libraries. Alpine leverages the packet capture library to multiplex the protocol flows ensuring consistency.

Trickle is used for ad-hoc traffic shaping such as the rate restrictions and prioritized transferring on TCP connections. That is, Trickle works as a session layer service. The main functionality of Trickle is implemented as a shared library, and it is transparently inserted into the application process image by the library preloading technique. The utility program automatically configures the preloading environment.

DR-TCP only provides downloadable and reconfigurable mechanisms for the traditional TCP protocol. DR-TCP is offered as a shared library to directly incorporate its mechanism into the application process. Unlike other systems, the library provides state machine objects instead of library functions. A customized loader program is offered to load the library into a specified memory area and appropriately binds the object symbols.

Table 6.2 presents a comparison of the network service features. The **protocol ranges** item shows how widely the protocol layers are supported as network services, the **composition** item is how flexibly and easily multiple network services are combined into the application, and the **Ad-hoc use** item expresses whether a network service can be instantly used.

Table 6.2: Comparison of network service features

System	Protocol ranges	Composition		Ad-hoc use	
General-purpose systems					
FreeNA	• Transport to Application	√	• Multi service • Automatic • Transparent	√	• Library style • No compiling/linking
Meta-Sockets	• Session to Application[1]	L	• Multi service • Programmatic • Transparent	L	• Java object style • Programmatic installation
TESLA	• Session to Application[1]	L	• Multiservice • Semi-automatic • Transparent	L	• C++ object style • Static link to the driver process[3]
DIT OOLS	• Session to Application[1]	L	• Multiservice • Programmatic[2] • Transparent	√	• Library style • No compiling/linking
Inter-position agents	• Session to Application[1]	–	• Single service	√	• Executable style • No compiling/linking
VTL	• Data-link to Application	–	• Single service	√	• Executable style • No compiling/linking
Single-purpose systems					
Alpine	• Transport	–	• Single service	√	• Library style • No compiling/linking
Trickle	• Session • Over TCP	–	• Single service	√	• Library style • No compiling/linking
DR-TCP	• Transport	–	• Single service	√	• Library style • Automatic linking

[1] Considering the internal implementation, the range of services could be feasible.
[2] An additional mechanism that combines arbitrary two service libraries are required.
[3] Flow handler classes has to be linked to the master process.

6.1.3 Reviewing from Applications' Perspective

The implementation characteristics of applications can restrict the network services to be inserted. For example, some systems leverages the language-inherent capability for extension, and using this method does not help to extend applications implemented in

other languages. Therefore, the difference in service insertion mechanisms influences the available application environment.

In addition, applications may use network functions in diverse ways, such as multiple communication flows, a shared communication flow, and asynchronous data communication. It is practical to say that network services are applied to only appropriate communication flows not every flow.

FreeNA's service insertion is directly applied to an application's process image, rather than a runtime system or the source code. The process image structure is not affected by the programming language used to implement the application. Some language systems compile applications as intermediate codes to work on their own virtual machines, like the Java language. Although FreeNA cannot handle such intermediate program directly, an interface between the VM and the underlying platform will be accessed. A notable aspect of FreeNA-enabled applications is that any source code or special compilation process is not necessary.

The control library offered by FreeNA manages every communication flow expressed as sockets. The address information, port number, transport-layer protocol, and application type (client or server) are registered for each socket, and the control library can use these information to apply different services by their flow types. Network service developers can implement their libraries as intermediate server processes to handle multiple flows across different processes for system-wide network services like congestion control.

MetaSockets uses the Adaptive Java language to introduce adaptability to traditional Java applications. Since Adaptive Java leverages the reflection mechanism to instantiate the MetaSocket classes, applications must access Java-implemented socket classes to insert filters. Filters are dynamically composed as a pipeline within the `send()` or `recv()` socket method according to the DM's decision. The DM considers the communication status changes such as the error rate, rather than the communication flow types.

The flow handlers for TESLA work as independent processes (master processes) from the application process, and the `tesla` wrapper program mediates between these processes

by delegating the socket-calls from one process to another. Therefore, any implementation form of application is available as long as it invokes the native socket-calls. Each master process can be shared by different processes, which allows for the comprehensive control of the network functions.

The DI runtime of DITOOLS handles the symbol table within the application's process image, and a module symbol and function symbol pair is manipulated. That is, applications utilizing native dynamic-link features can be extended by DITOOLS. The manipulation of the symbols is done once at the time of loading, and DITOOLS does not provide a manipulation mechanism during application execution.

Interposition agents can be loaded within the application process image directly by the Mach OS's features, and interpose an arbitrary user code into the specified system-calls. However, since agents are driven by the process-oriented approach, they cannot alter their behaviors based on the to application data.

Since VTL manipulates already transmitted packets at the interface between the guest OS and VMM, any application running on the guest OS is available with VTL. This packet manipulation is based on the libpcap/Winpcap library, which implies that VTL enables a conditional service insertion by using the header information, such as the IP address, port number, and protocol types.

Alpine's features are inserted into the application by the library preloading technique. Therefore, most applications are available without any modification. Since Alpine only provides a transport-layer service, the socket interface is simply overridden.

Trickle can work with applications that were implemented in many languages by using the library preloading technique. Trickle provides a common server program that coordinates among multiple trickle-enabled applications to globally limit the aggregated transfer rate.

Since DR-TCP is implemented as the shared library and loaded into the application process by the dedicated loader, most applications are available on this TCP protocol. Moreover, DR-TCP enables the application to replace the protocol implementation when

the environment is changed.

Table 6.3 presents a comparison of the adaptability to applications. The **implementation independence** item presents how independent the system is from the implementation style of the applications. The **insertion flexibility** item is how flexibly the service insertion types are supported by the system.

Table 6.3: Comparison of adaptability for application's features

General-purpose systems				
System	**Implementation independence**		**Insertion flexibility**	
FreeNA	√	• Any language • Binary-based extension • No recompilation/relinking	√	• Conditional insertion • Adaptable insertion • Cross-application insertion
MetaSockets	–	• Java language • Byte code-based extension • Linking to MetaSockets	√	• Adaptable insertion • Runtime reinsertion
TESLA	√	• Any language • Binary-based extension • No recompilation/linking	L	• Fixed insertion • Cross-application insertion
DITOOLS	√	• Any language • Binary-based extension • No recompilation/linking	–	• Fixed insertion
Interposition agents	√	• Any language • Binary-based extension • No recompilation/linking	–	• Fixed insertion
VTL	√	• Any language • VM-based extension • No recompilation/linking	√	• Conditional insertion • Cross-application insertion
Single-purpose systems				
Alpine	√	• Any language • Binary-based extension • No recompilation/linking	–	• Fixed insertion
Trickle	√	• Any language • Binary-based extension • No recompilation/linking	L	• Cross-application insertion
DR-TCP	√	• Any language • Binary-based extension • No recompilation/linking	√	• Adaptable insertion • Runtime reinsertion

6.1.4 Reviewing from Platforms' Perspective

As described before, the introduced systems including FreeNA take advantage of the various insertion mechanisms. Generally, these mechanisms are based on the platform-intrinsic aspects like API and ABI.

FreeNA leverages the abstractiveness of the socket interface by hiding the interposed services from the application. Socket-calls are intercepted by altering the addresses of the socket-calls within the process image if the Dyninst API is used. Since Dyninst API supports multiple executable file formats including ELF[81] and portable executables for Windows[82], and socket-calls can be hooked on multiple platforms in the same way. If inoperable formats are adopted by the platform, other interposing techniques like a library can be used by replacing the *Interposer* component of FreeNA server to another.

MetaSockets absorbs features of existing Java classes by using a dedicated compiler, and then traditional Java objects are finally created. Therefore, MetaSockets can work with standard JVM implemented on various platforms.

The network services for TESLA work as independent processes, and connect to an application by using the internal UNIX-domain sockets communication. The `tesla` wrapper program intercepts socket-calls from the application by using the library preloading technique. TESLA itself is implemented for Unix variant systems, and the abstraction mechanism of platform-dependent functions is not supported.

The DI runtime of DITOOLS leverages the runtime linker and loader to manipulate the symbol table of the process image. Linker and loader are deeply associated with the platform, and therefore, the platforms or the version of these systems are restricted.

Even though interposition agents provide abstraction system interfaces like path names and file descriptors, they depend on the Mach OS's memory management, process management, and system-call redirection mechanisms for loading agents into the application process and system-call interposing.

Since VTL is applied to packet streams, the packet capturing technologies of the platform determine its independency is determined. Modern platforms have packet fil-

tering/redirection mechanisms such as the BSD packet filter[83] and Windows Packet filter[84], and VTL can leverage these mechanisms using the libpcap/winpcap libraries.

Alpine moves the entire protocol stack code to the user-space, which allows developers to incorporate the implemented protocols into the kernel in the future. Since the protocol stack cannot be independent from the kernel on many platforms, not only the protocol stack code but also some kernel features, including the timers and data structures, have to be moved to the user-space.

Trickle was designed to work on Unix-like platforms that provide POSIX interfaces, and the utility program can set the `LD_PRELOAD` environment variable for library preloading.

Since DR-TCP uses a custom memory allocator, it can only work on limited architectures. Moreover, some kernel functions were also modified for ensuring end-to-end TCP session identification.

Table 6.4 presents a comparison of the system independences from the platform. The **platform abstraction** item represents how the system abstracts the platform-dependent issues. The **target platform** item is the platforms on which the system works.

Table 6.4: Comparison of system independency from the platform

General-purpose systems			
System	**Platform abstraction**		**Target platforms**
FreeNA	√	• Java-based system • Replaceable native interposer • Dyninst API	• Unix variants[2] • Microsoft Windows
Meta-Sockets	√	• Java-based system • Adaptive Java	• Java VM
TESLA	L	• Library preloading • POSIX interface	• Unix variants
DIT-OOLS	–	• Symbol table modification	• IRIX OS
Inter-position agents	–	• Mach's memory management • Mach's process management • Mach's syscall redirection	• Mach OS (BSD system interface)
VTL	√	• Packet manipulation/injection • Libpcap/Winpcap	• Xen • VMWare
Single-purpose systems			
Alpine	–	• Library preloading • Moving kernel code to user space	• FreeBSD
Trickle	L	• Library preloading • POSIX interface	• Unix variants
DR-TCP	–	• Custom memory allocator • Kernel modification	• Linux

[1] Other interposing methods are also available for dyninst-unsupported platforms.

[2] Other platforms can be supported by just implementing suitable interposer component.

6.2 The Manageability of FreeNA

As discussed before, FreeNA consists of many components and leverages various software techniques, such as the system-call interposition. Therefore, it seems that FreeNA's architecture raises the threshold of the system. In this section, the manageability of the

FreeNA components is described separately, and compared with other systems.

The FreeNA client is implemented as a fully independent Java program for the user interface. Therefore, it can be managed in the same way as normal Java programs.

The FreeNA server consists of a Java-coded part and the independent shared library named the Interposer. The Java-coded part can also be managed like a normal Java program. Since the Interposer library just calls the Dyninst API for the interposition, it can be managed like a normal shared library.

Dyninst API has been developed and managed as a part of the Paradyn project at Maryland University. In the FreeNA system, only the Interposer library of the FreeNA server uses this API, and FreeNA users and network service developers do not need to know the existence of the API. Therefore, we can integrate the latest API into the FreeNA system regardless of the users and service developers.

Since network service libraries are implemented as independent shared libraries with our-defined interface, and they are dynamically and fully-transparently loaded into the process image, and FreeNA users and service library developers do not need to always be the same. Therefore, FreeNA users can download service libraries from third parties, and manage them independently from the FreeNA client/server.

There are many other systems that leverage tricky interposition techniques. FUSE[26] hooks the file operation system calls to construct user-space filesystems. Although FUSE is integrated with the most recent Linux kernel and many distributors support it by default, the manageability of FreeNA is easy in that users can manage FreeNA in a unified way on many platforms, and all FreeNA's components run within the user-space as independent modules.

Xen[45] is another system that uses a binary rewriting method. In contrast, FreeNA users do not need to manage the patched version of applications, the modification of existing systems, nor do they need special CPU support.

6.3 Security

In this section, the software security of FreeNA is evaluated in view of the access control mechanism and vulnerability. The access control mechanism prevents FreeNA users from accessing other process information and OS resources via FreeNA's hack mechanisms. Moreover, FreeNA must have a protection mechanism for exploiting the process manipulation or system-call interposition to prevent malicious codes from taking control of the process.

6.3.1 Access control mechanism

FreeNA provides the client program for not only the system usability, but also security reasons. The separation of the FreeNA server and FreeNA client increases the level of system security of FreeNA because the FreeNA server can only manipulate the application processes under the internally defined operational commands specified by the client. Therefore, FreeNA users cannot exploit FreeNA's features to control the process behaviors.

To prevent application extension by unauthorized users, the FreeNA server forces an authentication process to the FreeNA client at the session establishment time. However, the simple user authentication overlooks the fact that authenticated FreeNA users can access any application because the FreeNA server runs in a privileged mode. One possible solution to this problem is that the FreeNA server could checks the permissions of the requested application with the native OS, and launch the application under the user privilege by using the `setuid()` or `CreateProcessAsUser()` system-calls.

6.3.2 Vulnerability

FreeNA allows users to download network service libraries from third parties, which can be harmful if an adversary covertly embeds malicious codes into the service library. However, FreeNA does not provide a protection method for removing malicious libraries. To solve this problem, FreeNA users should be forced to download from trusted servers

only, and a digital signature or checksum should be used to validate the downloaded libraries. In addition, users can statically investigate the system-calls invoked by the downloaded library using the binary check tools such as GNU Binutils[85] including `readelf` and `objdump` for Linux platform, and dumpbin[86] for Windows platform [1].

Generally, only privileged processes can open the raw socket. FreeNA gives the protocol server independence from the application process to run in privileged mode. Considering the protocol server only executes protocol processing, the risk of program hijacking is smaller than for other server systems, such as a Web server executing CGI or a database server executing SQL. To increase the server security, a sandboxing environment[27] can be applied to the protocol server. On the other hand, since the application process runs in user mode, it is possible to reduce the damage when the application is attacked.

6.4 Performance Evaluation

In this section, the performance overhead of the service insertion by FreeNA is evaluated. Figure 6.1 and Table 6.5 present the experimental network and each machine's specifications. In the experiments, the performance overhead of the service insertion was evaluated on both the Linux and Windows operating systems by comparing the throughputs of FreeNA-enabled applications to the throughputs of applications where the same services are directly implemented within the applications. The test applications were written in the C/C++ languages, and compiled by GNU g++ or Visual Studio 2008 using the best optimization option.

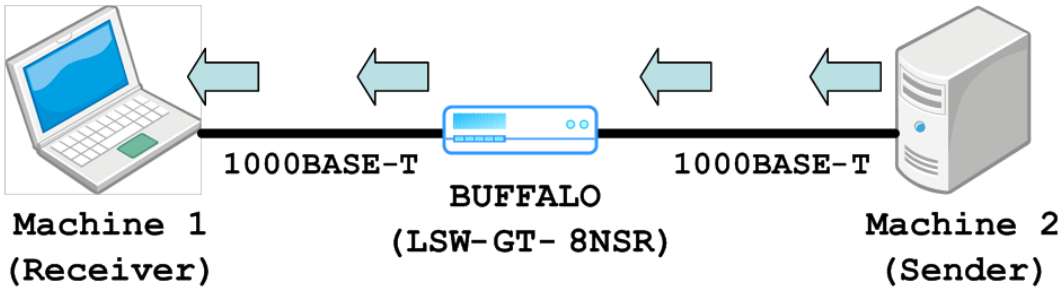

Figure 6.1: Experimental Network

[1]Many network services do not require OS support, especially, writing system-calls

Table 6.5: Machine specifications

Machine1	
OS	WindowsXP/Linux(2.6.18)
CPU	Intel PentiumM 2.13GHz
Memory	512 MByte
Ethernet	1000BASE-T
NIC Chipset	BCM95751M
Bus interface	PCI Express 1.0
HW Checksum	ON
Jumbo frame	OFF
Machine2	
OS	WindowsXP/Linux(2.6.18)
CPU	Intel PentiumD 2.8GHz
Memory	4 GByte
Ethernet	1000BASE-T
NIC Chipset	Yukon 88E8053
Bus interface	PCI Express 1.0
HW Checksum	ON
Jumbo frame	OFF

6.4.1 Performances of Upper-Layer Service Insertion

FreeNA-enabled Socket-call Overhead

First, the socket-call overhead with FreeNA was evaluated for machine 2. In the experiment, the actual execution time of the `send()` and `recv()` socket-calls were evaluated separately using one service library. The execution time was measured using the `RDTSC` instruction[87] of the x86 architecture. In addition, another application that statically calls a service function was also evaluated for comparison to the FreeNA-enabled application.

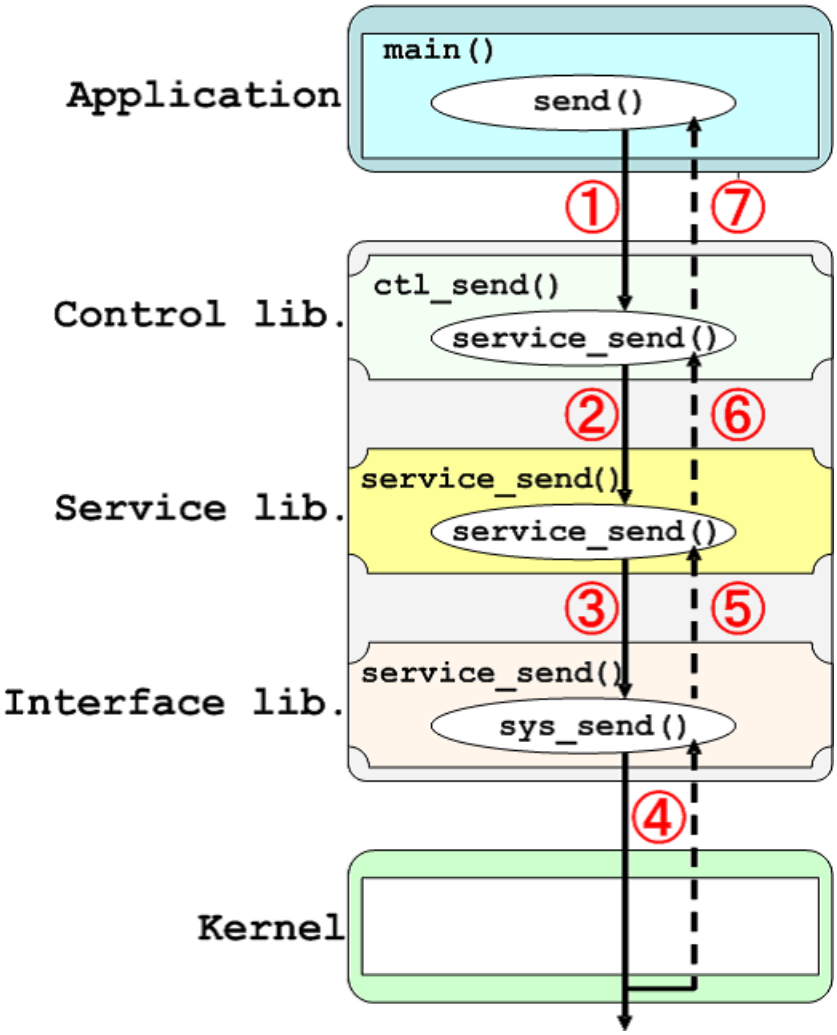

Figure 6.2: System-call overhead measurement points for the FreeNA-enabled application

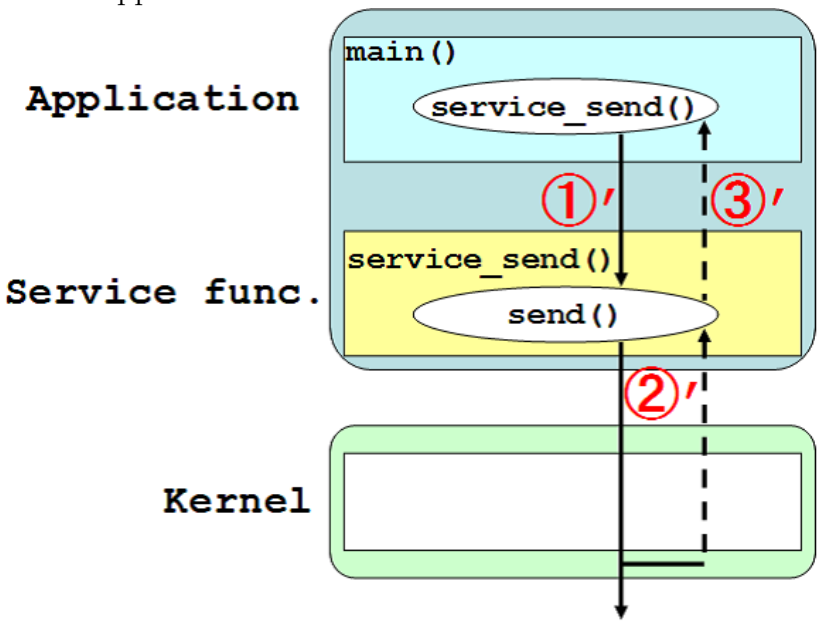

Figure 6.3: System-call overhead measurement points for the comparing application

Figures 6.2 and 6.3 show the execution time measurement points. While the FreeNA-enabled application invokes four functions, the other application only calls two functions per socket-call because the service function is directly implemented within the application. The `RDTSC` instruction was performed before/after the function calls.

Table 6.6: Sending socket-call overhead on Linux

FreeNA-enabled			Statically-implemented		
No.	Clock tick	μ sec	No.	Clock tick	μ sec
(1)	1844	0.659	(1)'	121	0.043
(2)	617	0.221	(2)'	21598	7.716
(3)	734	0.262	(3)'	108	0.039
(4)	23503	8.397			
(5)	146	0.052			
(6)	155	0.055			
(7)	222	0.079			
Total	27221	9.725	Total	21828	7.798
Total-(4)	3718	1.328	Total-(2)'	229	0.082

Table 6.7: Receiving socket-call overhead on Linux

FreeNA-enabled			Statically-implemented		
No.	Clock tick	μ sec	No.	Clock tick	μ sec
(1)	1969	0.704	(1)'	113	0.040
(2)	595	0.213	(2)'	11150	3.984
(3)	467	0.167	(3)'	104	0.037
(4)	11578	4.136			
(5)	110	0.039			
(6)	104	0.037			
(7)	238	0.085			
Total	15062	5.381	Total	11366	4.061
Total-(4)	3484	1.245	Total-(2)'	216	0.077

Table 6.8: Sending socket-call overhead on Windows

FreeNA-enabled			Statically-implemented		
No.	Clock tick	μ sec	No.	Clock tick	μ sec
(1)	3497	1.249	(1)'	1169	0.418
(2)	2332	0.833	(2)'	71203	25.439
(3)	1895	0.677	(3)'	911	0.326
(4)	59267	21.174			
(5)	730	0.261			
(6)	775	0.277			
(7)	754	0.269			
Total	69250	24.741	Total	73284	26.182
Total-(4)	9983	3.567	Total-(2)'	2080	0.743

Table 6.9: Receiving socket-call overhead on Windows

FreeNA-enabled			Statically-implemented		
No.	Clock tick	μ sec	No.	Clock tick	μ sec
(1)	2333	0.834	(1)'	965	0.345
(2)	1545	0.552	(2)'	23471	8.385
(3)	1377	0.492	(3)'	721	0.258
(4)	16999	6.073			
(5)	707	0.253			
(6)	708	0.253			
(7)	675	0.241			
Total	24344	8.697	Total	25157	8.988
Total-(4)	7345	2.624	Total-(2)'	1687	0.603

The performance overhead results are listed in Tables 6.8 and 6.9. From the results that the number of function calls affected the socket-call execution time at a microsecond order. However, since this experiment does not consider the heavyweight tasks, such as packet copying, data manipulation, and blocking for transmitting/receiving, the overhead of the socket-call execution with FreeNA will not impact on the overall application throughput.

Transmission Overhead with Light-weight Service

The next experiment evaluates the overall application throughput by continuously transmitting data packets without intervals. Since continuous data transmission can incur process blocking, the overhead of the socket-call execution with FreeNA will have less impact on the overall performance. In this experiment, the application on Machine 1 transmits 300,000 application-packets with a **Null** service library, which sends data simply

without any data manipulation or copying. Each application-packet is 1024 bytes and the time is measured at the application-level while all the data is transmitted to the receiver.

Table 6.10: Transmission time with light-weight service

Number of services		1	2	3	4	5
Linux	FreeNA-enabled	2.635	2.638	2.635	2.637	2.64
	Statically-implemented	2.629	2.635	2.636	2.637	2.636
Windows	FreeNA-enabled	10.105	10.104	10.12	10.121	10.095
	Statically-implemented	10.105	10.076	10.122	10.097	10.141

Table 6.10 lists the time for transmissions on both Linux and Windows at various null service libraries. Although the transmission times are different for different OSs, the execution time for the FreeNA-enabled application and for the comparing application that calls the service directly were almost the same (performance degradation was less than 2% at the most). Therefore, the overhead of a service insertion by FreeNA is negligible when the application continuously transmits application-packets.

Transmission Overhead with Heavyweight Service

This experiment was conducted under the same conditions as the previous one except that a **cryptography** service library and a **compression** service library were used. The compression library uses a *Zlib* library[74] and the cryptography library uses a *Crypto++* library[88], and a *Sosemanuk* stream cipher[89] was used in the experiment. We tested the cryptography services, compression services, and both the compression and cryptography services.

Table 6.11: Transmission time with heavy-weight services

Service Name		Cryptography	Compression	Crypto+Comp.
Linux	FreeNA-enabled	2.641	26.242	26.785
	Statically-implemented	2.641	26.197	26.721
Windows	FreeNA-enabled	10.404	23.574	24.141
	Statically-implemented	10.389	23.625	24.128

The measurement results are presented in Table 6.11 and indicate that FreeNA does not affect the performance of the target application when using practical service libraries

(performance degradation was less than 1%).

Overhead with practical usage

The third experiment was conducted using a file transfer application like FTP that uses two connections. A control connection was used to request a file and a data connection was used to transfer the file itself. The cryptography and compression service libraries were used again, and the compression service was applied to the data connection and the cryptography service was applied to both the data and the control connections. We evaluated the throughput of the client application on Machine 1 while the required file was transferring through the data connection using various application-data sizes. The size of the application-data was one of the important factors affecting the system throughput, because the number of packets is inversely proportional to their size. The number of packets should generally be reduced to curtail the overheads of the packet processing.

Figures 6.4 and 6.5 show the throughputs of the receiver-side applications on Linux. Figures 6.6 and 6.7 show the throughputs on Windows. In Fig. 6.4, the throughputs rapidly increased as the application-data size increased, especially for *Application-Normal*. The throughputs for these three schemes were close when the size was larger than 2048 bytes. As Fig. 6.5 shows, if the compression service was used, the throughput was low even though the actual application-data size was reduced by the compression. However, the throughput in the four graphs was almost the same. This is because the software-level compression takes up a lot of execution time, and therefore, the overhead of the service insertion by FreeNA has no impact on the overall throughput.

Next, on Windows, the throughput of the normal/cryptography clients was less than that of Linux. In addition, there are some substantial performance drop points including those of a normal application at data sizes of 2048 bytes, 4096 bytes, and 8192 bytes. A similar phenomenon can be seen in Gray et al.'s research[90]. According to Gray et al., *the network card driver's interrupt moderator feature throttles the interrupt requests when they are being issued at too fast a rate. At lower MTU settings with higher bus speeds, the number of interrupts thrown by the card would reach the interrupt moderator*

threshold faster, causing a wide variance in the throughput. However, the question is that why this phenomenon was not seen in the experiments on Linux despite the fact that the hardware architecture and software architectures of the applications, network services, and FreeNA are not changed. One possibility is to assume that the number of interrupts for notifying of the DMA completion may have been reduced on Linux by the NAPI just as Jin et al. pointed out[91]. Since the Linux NAPI lets the interrupt handler poll the DMA completion queue more frequently and handles more packets if there are some at any one time, the network card worked stably. Therefore, further research on the internal behavior of the network card and kernel protocol stack is needed to clarify this phenomenon.

As a whole, there were no significant differences between the FreeNA method and application-direct method. From the experiment, we can say that there seems to be no significant influence of the service insertion by FreeNA for practical usage on both platforms. Therefore, users only have to consider the overhead of the processing of the service functions themselves.

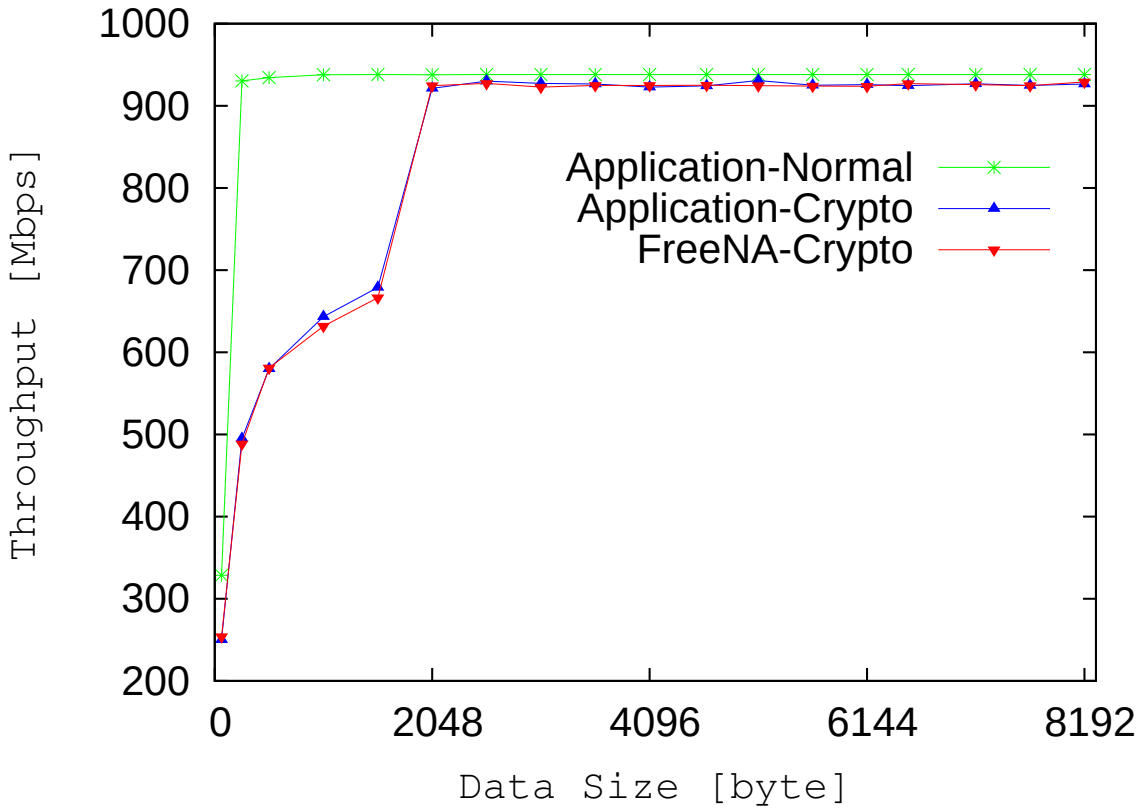

Figure 6.4: Throughput of client on Linux

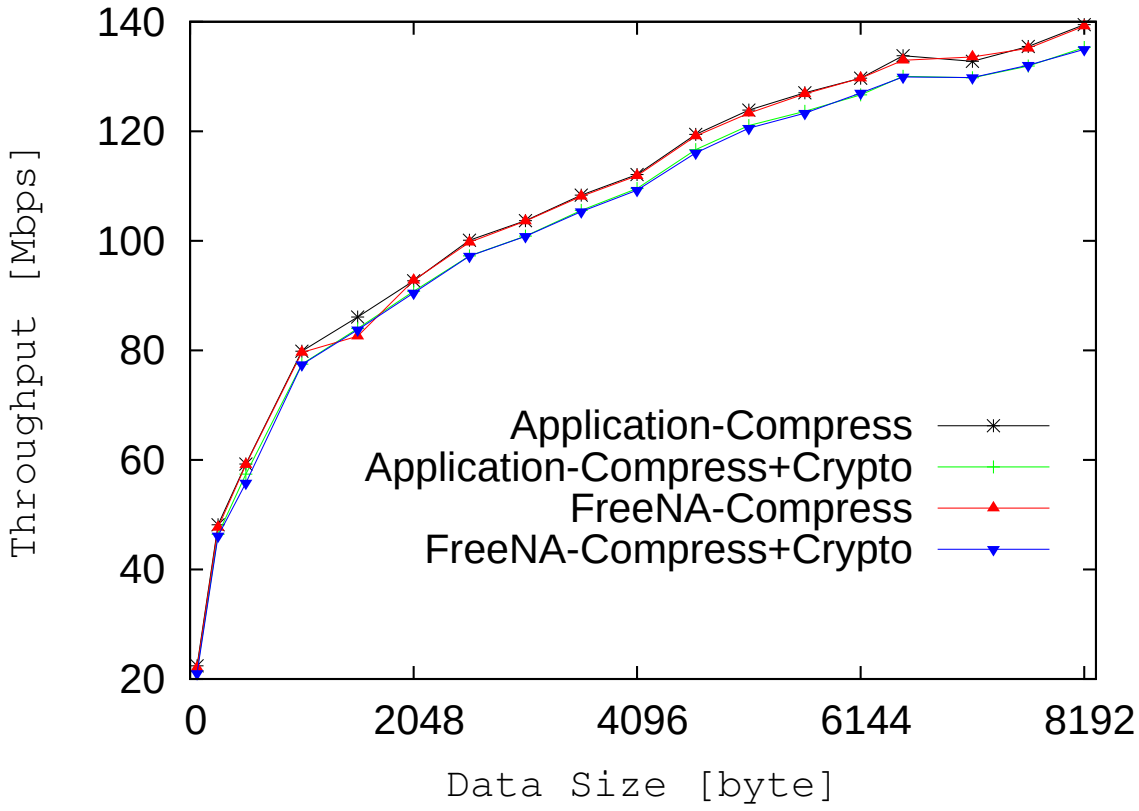

Figure 6.5: Throughput of client on Linux

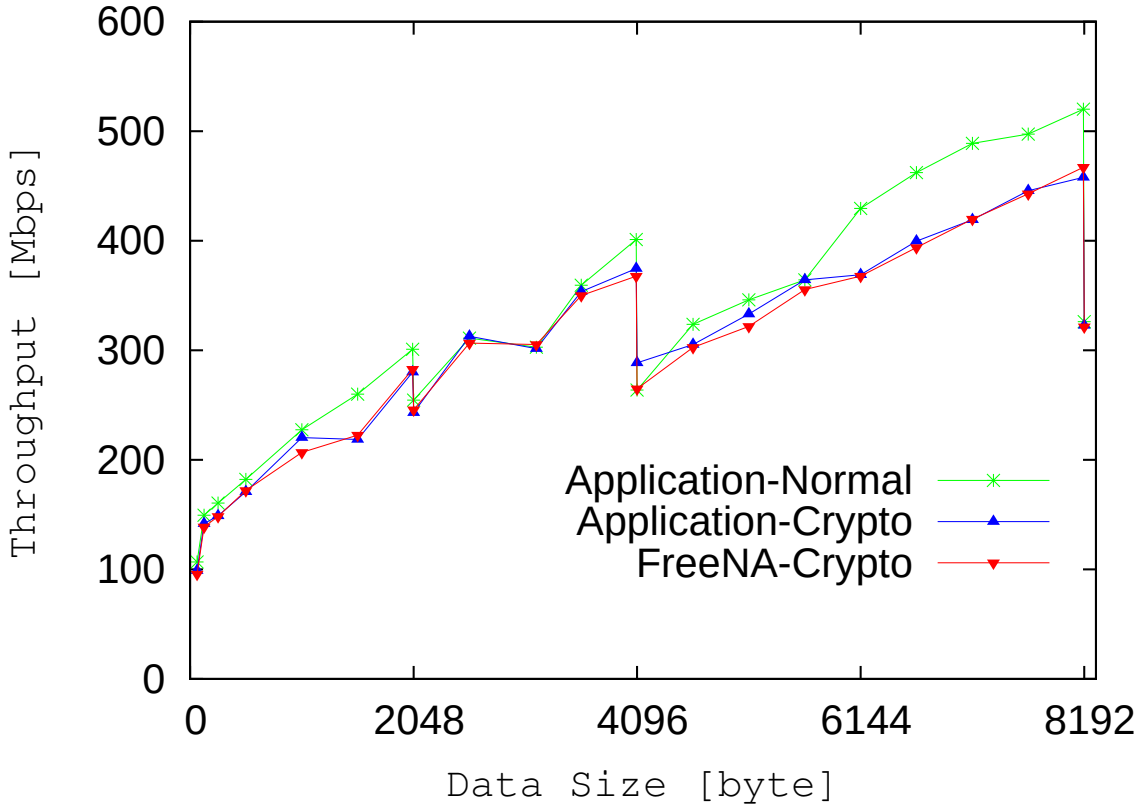

Figure 6.6: Throughput of client on Windows

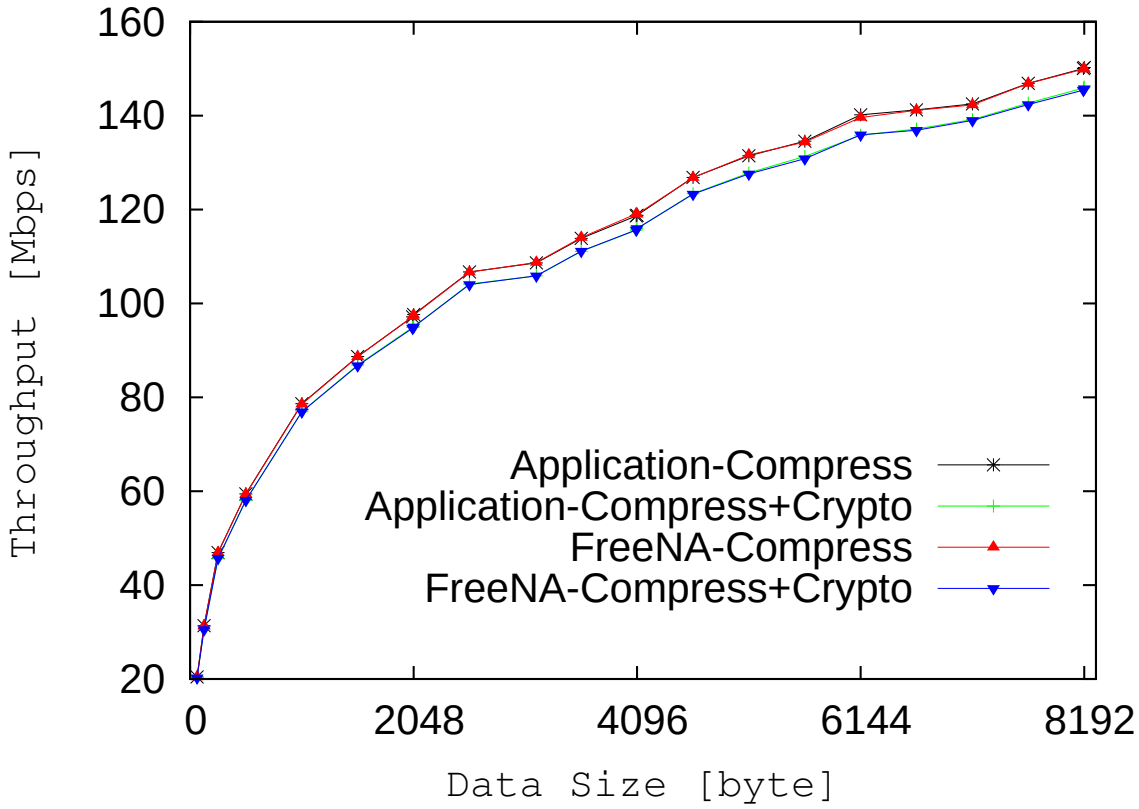

Figure 6.7: Throughput of client on Windows

6.4.2 Performances of Transport-Layer Protocol Insertion

Evaluation of transport-layer protocol insertion

To evaluate the overhead of a transport-layer protocol insertion, another UDP protocol stack consisting of the protocol server/library was implemented. As described in section 5.4, the protocol server executes the core functions of the protocol as independent processes, and the library intermediates between the server and the application process. In the experiment, this UDP implementation was used to compare the traditional UDP protocols within the kernel-space protocol stack. A test application sends or receives 1,000,000 consecutive packets at a specified data packet size, and the throughput was measured in the application.

Figure 6.8 shows the effective throughputs of the sender test application (S) and receiver test application (R) on Linux, and Fig. 6.9 shows the throughputs on Windows. From the results on Linux, we could see that there was no significant performance difference between the *UDP-FreeNA* and *UDP-Kernel* when the data size was small. This is because that the overhead comes from the packet processing and transferring within the OS, which had a relatively big impact. The effective throughput rose as the data size got larger, and finally peaked at 900Mbit/s because of the bandwidth limitation. When comparing the both the *UDP-FreeNA* and *UDP-Kernel* throughputs, the performance degradation was a maximum of 25% when the data size was 512 Bytes.

On the other hand, it is obvious that the overall throughputs on Windows are relatively less than those on Linux, and a significant performance degradation to *UDP-Kernel* was not observed, since the packet processing within the Windows kernel was inefficient compared to that of the Linux kernel and the overhead that came from the protocol insertion was negligible.

In addition, there were some substantial performance drop points when the data size was around 1024 bytes. This phenomenon was reported on the Usenet[92] as a small datagram transmission style on Windows that can cause burst packet loss on the receiver side. In practice, Windows uses a fast I/O path when the datagram is smaller than

the threshold size specified by the `FastSendDatagramThreshold` registry[2], and transmits a couple of datagrams to the NIC at a time. As a result, the receiver side NIC cannot handle the received packets when the transmission rate is too fast. While Windows cannot receive bursts of small packets, Linux has a NAPI mechanism that enables an interruption handler to take multiple packets from the NIC at a time. This is the reason why this phenomenon was not seen on Linux.

[2]Default value is 1024 byte[93].

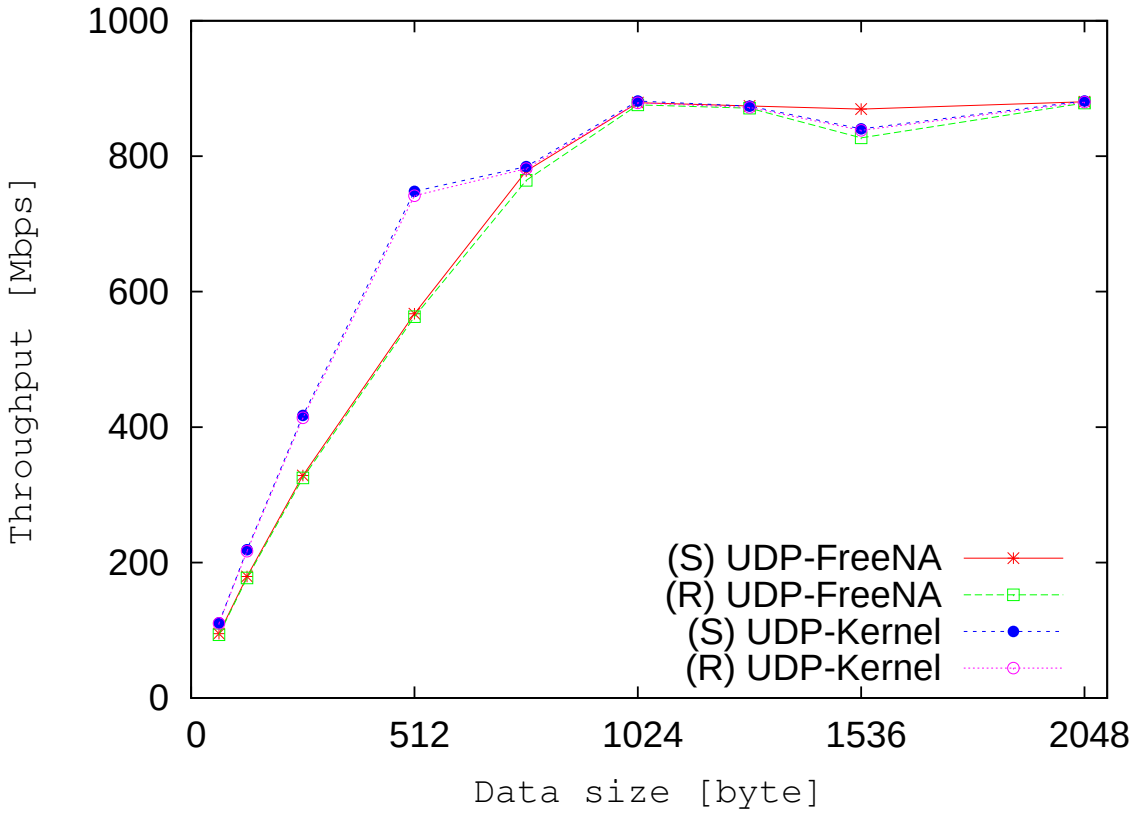

Figure 6.8: Throughput of the receiver application on Linux

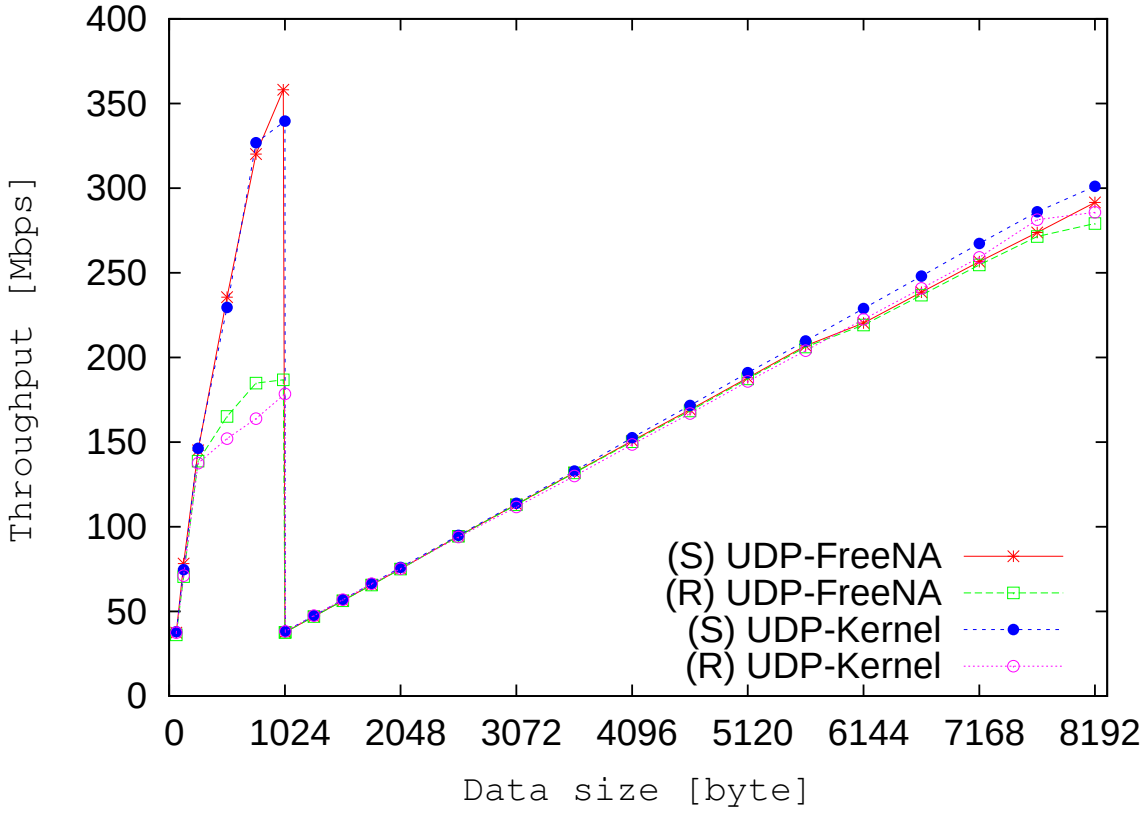

Figure 6.9: Throughput of the receiver application on Windows

Performances of dual communication flows

Figure 6.10 shows the results for the performance evaluation of the dual communication flows. In this experiment, two receiver applications were executed simultaneously on Linux. When comparing the results shown in Fig. 6.8, a pair of sender and receiver applications were executed, and each throughput of the receiver application decreased to one-half. In addition, there was no significant difference between the throughputs of *UDP-FreeNA* and *UDP-Kernel*. These results show that the protocol server of FreeNA treats multiple communication flows fairly.

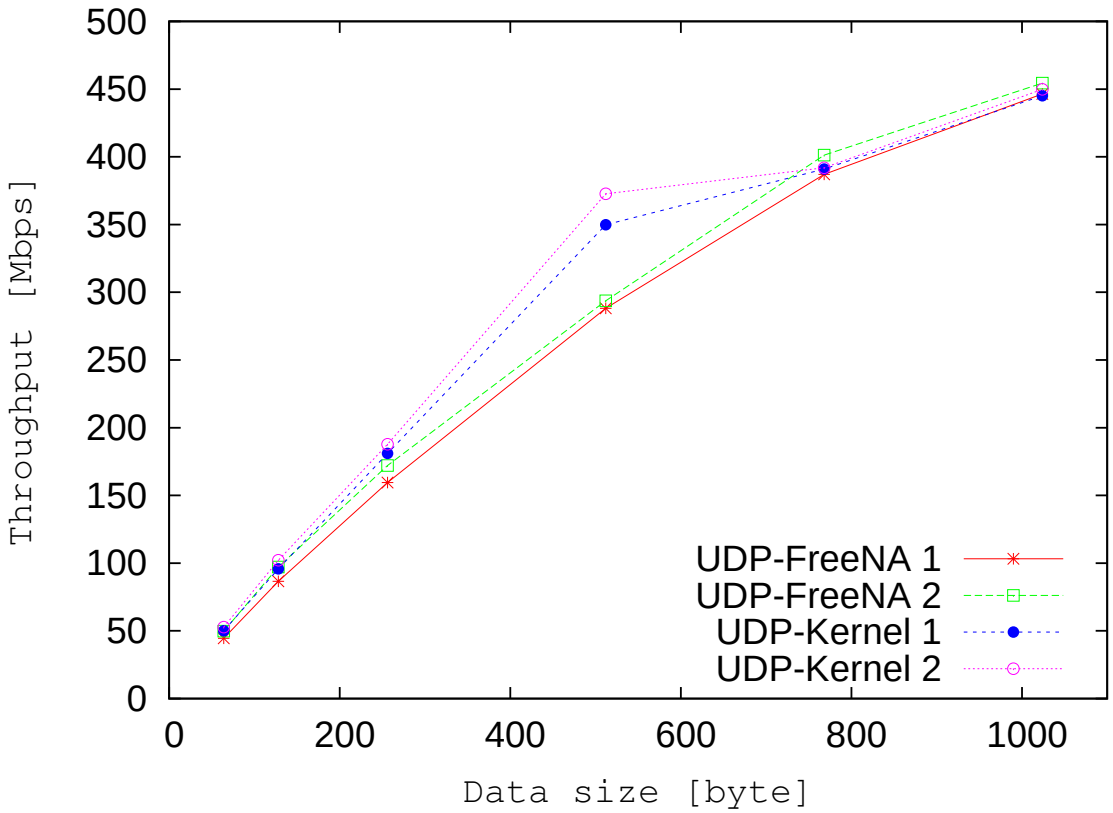

Figure 6.10: Throughputs of the two receiver applications running simultaneously

Discussion

As a result of the experiments, there was obviously some overhead when adding the protocol using the framework. First, I believe that this overhead came from the insertion of multiple libraries by the framework, as shown in Fig. 5.18. However, previous experiments

showed that the overhead of the library insertion was negligible.

Another reason would be that this performance degradation was assumed to be that there was frequent process switching between the application and the protocol server. That is, since the application process and server process communicate with each other as a one-way IPC using shared memory, the bounded buffer problem was caused. When the sender application is writing data packets into the shared memory, the protocol server was waiting. Therefore, the packet transfer rate decreases at that time.

Then, the CPU load average of communication time was evaluated on Linux (Fig. 6.11). The results showed that the load average exceeded 1 and frequently became 2 frequently with FreeNA, while the load average was constantly 1 when the UDP protocol within the kernel was used. This implies that the two processes competed for CPU resources during the communication, and the packet transfer rate decreased.

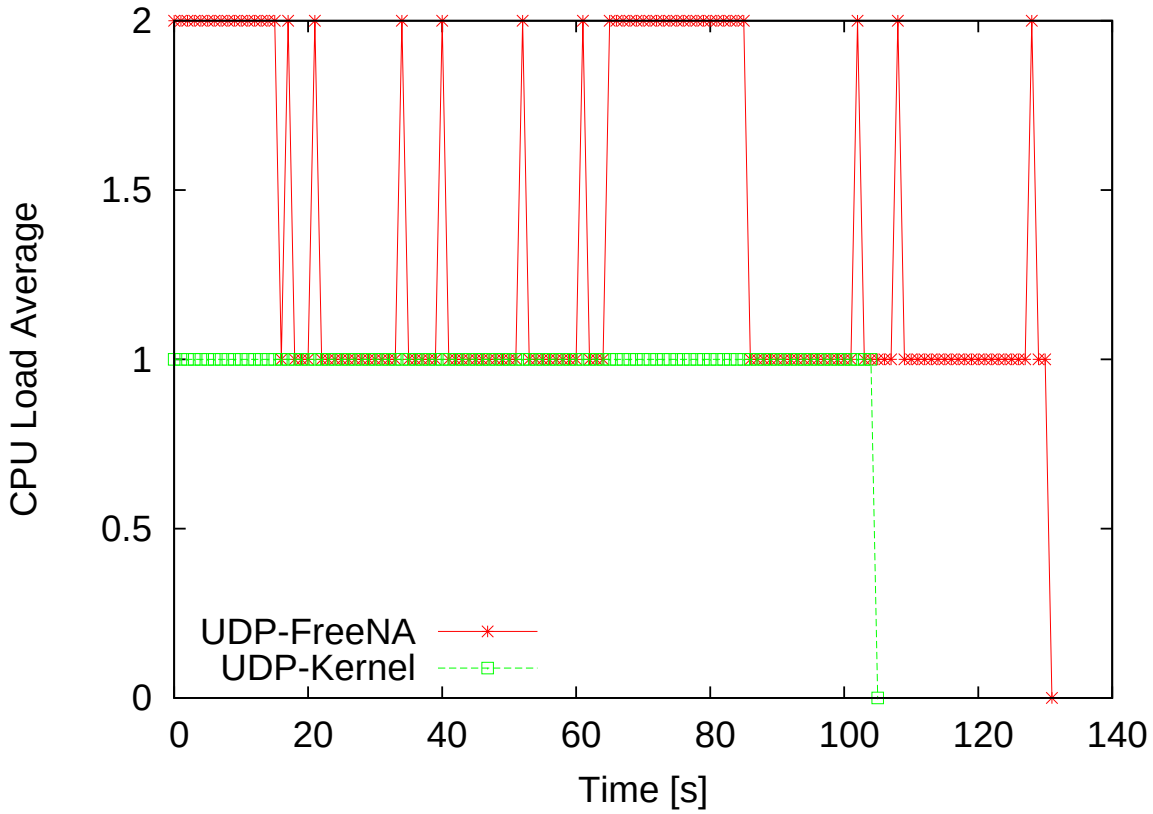

Figure 6.11: CPU load average during the communication

6.5 Summary

In this chapter, the FreeNA system was evaluated from the perspective of its functionality, manageability, security, and performance overhead.

The functionality of FreeNA was compared with other similar systems in view of its users, network services, applications, and platforms.

Since FreeNA is designed for both professional users and inexperienced users, the usability of FreeNA was evaluated by focusing on the configuration mechanism and system operability. Compared with similar systems, FreeNA provides a detailed and programless configuration mechanism, and also a user-oriented operation system to its users.

Each network service for FreeNA is implemented in view of the separation of concerns, flexible composition, and service deployment. As a result, the users can integrate the desired functionality gathered from various repositories into their applications without difficulty.

Generally, the implementation characteristics of applications can restrict the network services to be inserted. However, FreeNA is designed to handle applications written in various programming languages, as well as applications that use network functions in diverse ways.

FreeNA leverages the abstractiveness of a socket interface by hiding the interposed services from the application like in similar systems. However, FreeNA utilizes an abstract interposing mechanism by design and implementation, instead of directly using platform-dependent mechanisms. As a result, FreeNA can work on multiple platforms in the same manner.

Considering FreeNA is implemented as a comprehensive system, the manageability and security are also discussed.

To extend existing applications without directly modifying their source codes, the application-level performance will decrease because of the additional overhead. Although the relationship between the service insertion performance and insertion flexibility is a trade-off, FreeNA enables for flexible service insertions with relatively lower performance

degradation compared with other systems. In practice, the insertion of upper-layer services requires less than 2% of a throughput degradation, and a 25% degradation when inserting transport-layer services.

Chapter 7

Conclusion

The purpose of this study is to investigate a practical methodology for promoting network systems' functionalities. As represented by the Internet, networks have been a social infrastructure that support real business activities, educational and academic activities, and many other activities. Existing network systems will continue to be extended to fulfill users' requirements. However, most existing network systems have been developed based on the traditional TCP/IP protocol suite, and it is difficult to introduce innovative network technologies because of their interoperability issues.

7.1 Contributions

In this paper, the mechanism of adaptive computing was focused on as a key technology for keeping up with the progress of networks. First, many practical techniques for introducing adaptability into existing systems were investigated, including their characteristics and pitfalls. Based on the investigation, the necessary nature of a system that can make existing systems adaptable was considered from the perspectives of technology and usability. In practice, the FreeNA framework was designed and implemented as a concrete adaptable communication system. According to the evaluation results, the proposed methods not only can achieve more users' requirements than other similar methods, but also showed considerable efficiency in the communications in a real environment.

The following are the main contributions of this study.

(a) A design and an implementation method was proposed for applying a variety of net-

work services into existing systems. FreeNA supports multi-layer network services including transport-layer protocols up to application-specific services. Transport-layer protocols for FreeNA-enabled applications are implemented in the user-space as independent executables using raw socket interfaces. The upper-layer services are directly incorporated into the application process as dynamic-loading libraries. An abstract design of the socket interface allows these network services to be utilized by the application in the same manner.

(b) A mechanism for flexibly composing the network functionality of applications was proposed. The flow handler concept allows for the flexible composition of network services at runtime by chaining them through the recursive socket interface. FreeNA utilizes yet another structure of the flow handler concept in order to achieve application-specific and efficient network services. In addition, the behavior of the network service can vary at runtime. That is, the network service can be applied to the specific communication flow when the application uses multiple flows, and conversely the network service can handle multiple communication flows across several applications.

(c) FreeNA was designed to have an affinity with the existing network systems. First, network services can be incorporated into existing applications without any modification using the socket-call interposition technique. Although there are many interposition techniques, FreeNA takes the binary-modification approach for multiple programming languages support, flexible modification, and better performance. Moreover, FreeNA can work on multiple-platforms in the same manner without modification of the platform by dividing the platform-dependent issues such as API, ABI, and interposition methods from the core system. In particular, it is necessary to emphasize that two major operating systems, Microsoft Windows and Linux, can use FreeNA's platform.

(d) The user-oriented approach for extending existing systems was proposed. Generally,

programmatic skills and technical knowledge are required to enhance the functionality of network systems. However, FreeNA allows inexperienced users to extend their systems without using the programmatic approach. FreeNA offers front-end systems that can be customized to their user-interfaces for easy system operation, and a XML-based configuration mechanism for functional designation of the network services. Since the network services themselves are implemented as independent components under the defined system interface, users can take advantage of the network services developed by third-parties.

(e) FreeNA achieved a low performance overhead when utilizing the network services incorporated into applications. Since most network services are directly inserted into the application process, their mechanisms can be used via a local function call. The performance evaluation results showed that the functions of the upper-layer services can be used with two percent of the throughput degradation at worst compared with the throughput of an application that is extended directly to have the corresponding network services. In addition, the transport-layer protocol inserted by FreeNA requires 25% of the performance overhead compared with the kernel-implemented protocol. Considering that the FreeNA-enabled application can achieve a good performance near the gigabit speed under a commodity system environment, FreeNA can fulfill the performance requirements of a practical network environment.

To summarize this paper for each chapter, in chapter 1, the problems of current network environments and a road map to make a better network environment including the necessary systems and users was described as the background for this paper. As the problems, the current network is rapidly evolving based on the traditional mechanisms and network systems have to keep up with this evolution without losing interoperability. The concept of autonomous computing and compositional adaptation are effective because network systems can be extended while maintaining the system compatibility without creating more troublesome works for users. Therefore, FreeNA was proposed as an instance for these concepts.

In chapter 3, the fundamental policies for FreeNA were discussed to create a general-purposed framework for adaptive communications and a usability system. To meet the various network service requirements, network systems should be flexibly and continually extended. FreeNA was designed as a flexible system where users can configure their systems' functionality using a user-oriented system interface. Therefore, the implementation form, system usage, network service implementation, platforms, and applications were discussed in view of a system where FreeNA can be used as an application/user-oriented network tool providing a mechanism to users for more easily customizing and combining services without needing platform and application support.

In chapter 4, the architectural design of FreeNA and the user experience were explained. The inserted network services were located in the communication path between end-to-end applications. To instantiate the network services, the concept of a flow handler and flow handler chain that compose the network service components independent of each other were introduced. The architecture of the FreeNA system was designed composing a user-friendly client system with a server system. To ensure the portability of FreeNA, the server divides its tasks into platform-independent tasks and platform-dependent tasks. In addition, the XML-formatted configuration file was provided taking into account inexperienced users by adopting a functional selection approach. This enables for a detailed configurability of the network functionality of FreeNA-enabled applications.

In chapter 5, a practical mechanism of the FreeNA system including the client/server systems, flow hander (chain) concept, transport-layer protocol insertion, and protocol-free environment were described. The interposer components, as the key components of the FreeNA server, were prepared for each platform in order to ensure the system portability. Each network service was implemented as a shared library, and combined with other service libraries by pointer-based abstraction. At runtime, FreeNA also inserts other network services to control the network service hierarchy and accesses platforms provided by the network functions. As the engine of service insertion, a system-call interposition based on a binary modification technique was leveraged. FreeNA also enables a transport-

layer protocol insertion using a raw IP mechanism, and a dedicated process was introduced to handle the communication flows across multiple processes. Moreover, the adaptive mechanism for switching the transport-layer protocol was implemented to maintain the network interoperability.

In chapter 6, the FreeNA system was evaluated from various perspectives of functionality and performance overhead. The functionality of FreeNA was compared with other similar systems in view of the users, network services, applications, and platforms. The advantages of FreeNA's functionality are as follows. FreeNA provides a detailed and programless configuration mechanism, and also a user-oriented operation mechanism to its users, while many other systems do not provide such mechanisms. With FreeNA, users can integrate the desired functionality gathered from various repositories into their applications without needing to perform complex tasks. FreeNA was designed to handle applications written in various programming languages, rather than depend on language-specific mechanisms. Moreover, a diversity of usages of the network communication flows by the applications were also supported. FreeNA can work on multiple platforms in the same manner by utilizing an abstract interposing mechanism. The results of the performance overhead showed that the overhead of a service insertion by FreeNA was relatively small when considering other insertion methods even though the relationship between the service insertion performance and the insertion flexibility is a trade-off.

7.2 Discussion

In this section, the aforementioned FreeNA's contributions are discussed from the validity point of view. Since many researchers and engineers have proposed similar systems or practical techniques, FreeNA should be compared with these methods.

(a) Many similar studies can be divided to service-specific systems and general-purposed systems. Service-specific systems, such as Trickle[24], TCP Stream Filtering[25], Alpine[56], and DR-TCP[57], have specialized their architectures and system usages to support only a limited functionality. Moreover, they do not provide a

more flexible configuration mechanism than FreeNA provides because of the application transparent requirement. Next, general-purposed systems, such as TESLA[23], MetaSockets[13], DITOOLS[22], and VTL[39], provide multi-functional service compositions. Other than VTL, they do not support cross-layer network services including transport-layer protocols. Even though VTL can handle lower-layer protocols by using direct packet manipulation under a VM environment, it requires a tremendous amount of tasks to provide the protocol service because they cannot take advantage of platform support. However, FreeNA can seamlessly handle cross-layer services as a bi-directional pipeline over the kernel protocol stack of the platform.

(b) As described above, general-purposed systems can support several types of network services. However, only a few systems can compose multiple network services independent of each other. TESLA uses the flow handler concept in its original way to achieve flexible service composition. However, the network services for TESLA have to be work separately from the application to handle the cross-application communication flows. This implementation style makes it difficult to realize the application-specific services that handle the process resources, and result in a heavy amount of performance overhead at runtime. On the other hand, FreeNA achieved a flexible service composition within the application process. Moreover FreeNA can combine the two service implementation types in the same service pipeline to handle multiple flows in across-the-board applications.

(c) System-call interposition has been one of the most widely used techniques for extending an application's functionality without needing to source code level. As discussed in Chapter 2, language-based techniques, binary-based techniques, OS-based techniques, VM-based techniques, and proxy-based techniques are available. Since each interposition technique has advantages and drawbacks respectively, it is undesirable to rely on a single technique. FreeNA introduces an abstract interface to the system-call interposition in order to divide the platform-dependent issues. Even though FreeNA utilizes a binary-based interposition using Dyninst API (This

is available on many OSes and CPU processors), the interposition mechanism can be replaced in accordance with the platform. Indeed, the introduced similar systems that have already been introduced do not support multiple platforms at the native level.

(d) Many similar systems do not provide user-oriented system usability because they suppose the system users and network service developers are on in the same. For example, MetaSockets provides adaptable socket classes to users to configure the network composition at the code-level, and TESLA requires a compilation process to compose network services. That is, many systems do not provide easy service compositions and system operation mechanisms, even though they support service insertion without modification of the application. Considering these systems are useful to inexperienced users, such as network administrators and application users, network service developers and users should be treated the same, and should be able to create an ad-hoc application enhancement mechanism.

(e) Generally, the flexibility of the system and the actual performance are in relation to the trade-off. Since TESLA is developed to completely divide applications and network services, the peak throughput of TESLA was decreased over 50 percent when using a lightweight service. VTL is also developed to support a wide range of network services on many platforms. However, since it has to capture the transferring packet stream at the interface between the VM and the host OS, the peak throughput was decreased to near 80%. Compared with these systems, FreeNA's performance is significantly superior even though FreeNA enables a flexible service composition.

7.3 Future Work

In this paper, the transparent approach to extend existing systems is achieved on major platforms. FreeNA provides a flexible network service composition mechanism to users in a user-oriented manner. However, since FreeNA is still a proof-of-concept framework, many

challenges still remain in order to achieve FreeNA's goals, which are dynamically adapted communications and autonomous computing for widespread network environments. In this section, the remaining challenges for promoting FreeNA's effectivity are described.

7.3.1 Full Fledged Deployment into Practical Environment

In practical systems, there are many obstacles for deploying the FreeNA system. For example, a strict security mechanism on a the platform like SELinux[94] that might restrict the modification of the applications and available resources, a firewall might block the communication that one's own services are used, and FreeNA might require complex set up processes for the entire FreeNA system for the users of some environments. Therefore, these practical issues related to the platform have to be investigated to make FreeNA cross-platform a friendlier system.

7.3.2 Full Autonomic Service Insertion

Even though FreeNA provides user-oriented system usability such as operational programs and configuration files, users have to recognize the existence of FreeNA, prepare the desired network service libraries in advance, and compose the services by writing the configuration file. Ideally, users should use the FreeNA system indirectly, and appropriate network services should be automatically downloaded from the network environment and inserted into the application dynamically like in DR-TCP[57] and the application-layer protocol conversion system[55].

7.3.3 Dynamic Updates of the Network Service Composition

The SDP's offer/answer model allows for an update mechanism of the session content after the negotiation. Although network services currently have to be composed by FreeNA before the communication, this offer/answer model would enable the dynamic recomposition of network services during a communication, like in MetaSockets. However, this will require specific definitions of the thresholds and conditions that trigger the renegotiation by the offer/answer process. In addition, data consistency between the

end-to-end applications must be ensured while replacing the network services or protocols.

Acknowledgment

本論文をまとめるにあたり，熱心にご指導して頂いた全ての先生方に対して謹んで感謝の意を表します．

特に，筆者の主任指導教官でいらっしゃいました計 宇生准教授には，３年間もの長きにわたって懇切なるご指導とご協力を頂きました．岩手県立大学の博士前期課程から進学してきた筆者を快く受け入れて下さり，さらには博士後期課程での研究テーマにつきましても，筆者の希望のままで自由に研究させて頂きました．また，本研究を進める上で，通信システムの専門家として有益な助言を何度も頂きました．この様に，先生の寛大かつ献身的なご指導がありましたからこそ，途中で挫けることなく博士後期課程での３年間を研究活動に注力することができました．厚く御礼申し上げます．

丸山勝巳教授には，総合研究大学院大学への進学が決定した時に大変有益な文献を紹介して頂きました．この文献なくしては本研究をこの様な形でまとめることはできませんでした。先生には，本研究が始まった当初から現在に至るまでの間，幾度となく貴重なご指摘を頂きました．先生のシステムソフトウェアに関する豊富な知識と経験のおかげで，多角的な視点で本研究を見つめることができ，提案したシステムもより洗練されたものにすることができました．真にありがとうございました．

本論文の審査員を担当して頂いた，国立情報学研究所の中村素典教授，山田茂樹教授，福田健介准教授，および岡山大学の谷口秀夫教授に厚く感謝の意を表します．論文審査過程における先生方の鋭いご指摘により，本論文をより質の高いものにすることができました．谷口先生におきましては，論文審査とは別に研究会においても度々筆者の研究に耳を傾けて下さり，有益なアドバイスもして頂きました。

そして，筆者が学術研究の世界へ足を踏み入れるきっかけを作って頂いた，学部・博士前期課程での指導教官である岩手県立大学の蔡 大維准教授，阿部芳彦教授にも感謝を

申し上げます．先生方のご指導のおかげでここまで来ることができました．

また，計研究室の皆様にも様々なご支援を頂きました．この場を借りてお礼を申し上げます．

Bibliography

[1] Hiroyuki Morikawa and Tomonori Aoyama. Realizing the Ubiquitous Network: the Internet and Beyond. *Telecommunication Systems*, Vol. 25, pp. 449–468, 2004.

[2] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, Vol. 36, No. 1, pp. 41–50, January 2003.

[3] Jr. G.H. Campbell. Adaptable Components. In *Proceedings of the 21st international conference on Software engineering*, pp. 685–686, 1999.

[4] G. T. Heineman. Adaptation of Software Components. In *2nd Annual Workshop on Component-Based Software Engineering*, 1999.

[5] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing Adaptive Software. *IEEE Computer*, Vol. 37, No. 7, pp. 56–64, June 2004.

[6] Ye Tian, Kai Xu, and Nirwan Ansari. TCP in Wireless Environments: Problems and Solutions. *IEEE Radio Communications*, pp. 27–32, March 2005.

[7] Saverio Mascolo, Claudio Casetti, Mario Gerla, M.Y. Sanadidi, and Ren Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *ACM Mobicom*, pp. 287–297, July 2001.

[8] Kai Xu, Ye Tian, and Nirwan Ansari. TCP-Jersey for Wireless IP Communications. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, Vol. 22, No. 4, pp. 747–756, May 2004.

[9] NIST. Data Encryption Standard (DES). *FIPS 46-3*, 1999.

[10] NIST. Advanced Encryption Standard (AES). Technical report, FIPS 197, 2001.

[11] Kiczales Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, Vol. 1241, pp. 220–242, 1997.

[12] Ji Zhang and Betty H.C. Cheng. Towards Re-engineering Legacy Systems for Assured Dynamic Adaptation. In *IEEE International Workshop on Modeling in Software Engineering*, 2007.

[13] S. M. Sadjadi, P. K. McKinley, E. P. Kasten, and Z. Zhou. MetaSockets: design and operation of runtime reconfigurable communication services. *Software Practice and Experience*, 2006.

[14] E. P. Kasten, P. K. McKinley, S. M. Sadjadi, and P. E. K. Stirewalt. Separating Introspection and Intercession to Support Metamorphic Distributed Systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCSW'02)*, 2002.

[15] E.P. Kasten and P.K. McKinley. Adaptive Java: Refractive and Transmutative Support for Adaptive Software. Technical report, Technical Report MSU-CSE-01-03, 2001.

[16] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, 2002.

[17] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns*. Addison-Wesley, 1995.

[18] Shigeru Chiba and Michiaki Tatsubori. Structural Reflection by Java Bytecode Instrumentation. *IPSJ Journal*, pp. 2752–2760, 2001.

[19] M.B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. *ACM SIGOPS Operating Systems Review*, pp. 80–93, 1993.

[20] Michael Blair Jones. *Transparently Interposing User Code at the System Interface*. PhD thesis, Carnegie Mellon University, 1992.

[21] The mach project home page. http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html.

[22] Albert Serra, Nacho Navarro, and Toni Cortes. DITOOLS: Application-level Support for Dynamic Extension and Flexible Composition. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 225–238, 2000.

[23] Jon Salz and Alex C. Snoeren. TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-end Network Services. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, 2003.

[24] M.A. Eriksen. Trickle: a userland bandwidth shaper for Unix-like systems. In *Proceedings of USENIX Annual Technical Conference*, 2005.

[25] K. Kono, T. Shinagawa, and M.R. Kabir. Improving Internet Server Security by Filtering on TCP Streams. *IPSJ Journal*, 2005.

[26] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/.

[27] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 293–306, 2008.

[28] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conference on PLDI*, 2007.

[29] B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 2000.

[30] Giridhar Ravipati, Andrew R. Bernat, Nate Rosenblum, Barton P. Miller, and Jeffrey K. Hollingworth. Toward the Deconstruction of Dyninst. Technical report, Computer Science Department, University of Wisconsin, 2007.

[31] livepatch – Live Patching for Linux. http://ukai.jp/Software/livepatch/.

[32] Binary File Descriptor Library manual. http://sourceware.org/binutils/docs/bfd/History.html.

[33] N.C. Hutchinson and L.L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transaction on Software Engineering*, 1991.

[34] Sean W. O'malley and Larry L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, Vol. 10, No. 2, pp. 110–143, May 1992.

[35] Norman C. Hutchinson and Larry L. Peterson. Design of the x-Kernel. *ACM SIGCOMM Computer Communication Review*, pp. 65–75, 1988.

[36] Dennis M. Ritchie. A Stream Input-Output System. *AT & T Bell Laboratories Technical Journal*, Vol. 63, No. 8, pp. 1897–1910, 1984.

[37] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, and Todd A. Proebsting. Scout: A Communications-Oriented Operating System. In *Fifth Workshop on Hot Topics in Operating Systems*, 1994.

[38] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. *ACM SIGOPS Operating Systems Review*, Vol. 30, pp. 153–167, 1996.

[39] John R. Lange and Peter A. Dinda. Transparent Network Services via a Virtual Traffic Layer for Virtual Machines. In *Proceedings of the 16th international symposium on High performance distributed computing*, pp. 23–32, 2007.

[40] The libpcap project. http://sourceforge.net/projects/libpcap/.

[41] WinPcap: The Windows Packet Capture Library. http://www.winpcap.org/.

[42] Libnet home page. http://libnet.sourceforge.net/.

[43] Prashanth P. Bungale and Chi-Keung Luk. PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation. 2007.

[44] Luk C. K., Cohn R. S., Muth R., Patil H., Klauser A., P. G. Lowney, Wallace S., Reddi V. J., and Hazelwood K. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *Pragramming Languages Design and Implementation 2005*, pp. 190–200, 2005.

[45] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles*, 2003.

[46] Neiger. Gil, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, pp. 167?–178, 2008.

[47] Michal Trojnara. stunnel - multiplatform SSL tunneling proxy. http://stunnel.mirt.net/.

[48] Zhenyun Zhuang, Tae-Young Chang, Raghupathy Sivakumar, and Aravind Velayutham. Application-Aware Acceleration for Wireless Data Networks: Design Elements and Prototype Implementation. *IEEE Transactions on Mobile Computing*, Vol. 8, No. 9, pp. 1280–1295, 2009.

[49] netfilter/iptables project homepage. http://www.netfilter.org/.

[50] Windows Filtering Platform. http://www.microsoft.com/whdc/device/network/wfp.mspx.

[51] Simone Tellini and Renzo Davoli. Design and Implementation of a Multifunction, Modular and Extensible Proxy Server. *Proceedings of the 4th International Conference on Networking*, 2005.

[52] Simone Tellini and Renzo Davoli. Prometeo internals. Technical report, Department of Computer Science, University of Bologna.

[53] xinetd. http://xinetd.org/.

[54] B. Carpenter. Internet Transparency, 2000.

[55] Masakuni Agetsuma, Kenji Kono, Hideya Iwasaki, and Takashi Masuda. Exploiting Mobile Codes for User-Transparent Distribution of Application-Level Protocols. *IEICE Journal on Information and Systems*, 2003.

[56] David Ely, Stefan Savage, and David Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, 2001.

[57] Jae-Hyun Hwang, Jin-Hee Choi, Se-Won Kim, and Chuck Yoo. DR-TCP: Downloadable and reconfigurable TCP. *ScienceDirect The Journal of Systems ans Software*, pp. 83–99, 2008.

[58] Yunhong Gu and Robert L. Grossman. Supporting Configurable Congestion Control in Data Transport Services. In *Proceedings of the 2005 ACM/IEEE SC—05 Conference*, 2005.

[59] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. *RFC 2960*, 2000.

[60] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). *RFC 4340*, 2006.

[61] OPNET - Making Networks and Applications Platform. http://www.opnet.com/.

[62] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[63] Jorn Justesen, Tom Hoholdt, (Shojiro Sakata, Masazumi Kurihara, Hajime Matsui, and Masaya Fujisawa in translation). *A Course In Error Correcting Codes*. Morikita Publishing Co., Ltd., 2005.

[64] Man-Keun Seo, Yo-Won Jeong, Jae-Kyoon Kim, and Kyu-Ho Park. A New Packet Loss-Resilient Duplicated Video Transmission. In *IEEE Asia-Pacific Coference on Communications*, 2005.

[65] IP_DUMMYNET. http://info.iet.unipi.it/luigi/ip_dummynet/.

[66] The Zorp gateway technology. http://www.balabit.com/network-security/zorp-gateway/.

[67] National Institute of Standards and Technology. http://www.nist.gov/index.html.

[68] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. *RFC 2401*, 1998.

[69] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. *RFC 5246*, 2008.

[70] Wi-Fi Alliance. http://www.wi-fi.org/.

[71] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.

[72] OpenSSL - The Open Source toolkit for SSL/TSL. http://www.openssl.org/.

[73] libcurl - the multiprotocol file transfer library. http://curl.haxx.se/libcurl/.

[74] Greg Roelofs and Mark Adler. zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library. http://www.zlib.net/.

[75] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. *RFC3261*, 2002.

[76] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. *RFC4566*, 2006.

[77] Jon Postel. Internet Protocol. *RFC 791*, 1981.

[78] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with the Session Description Protocol (SDP). *RFC 3264*, 2002.

[79] W.Richard Stevens (and Yoichi Shinoda in translation). *UNIX Network Programming*, Vol. 1. Pearson Education Japan, 2003.

[80] A. I. Sundararaj and P. A. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. In *In Proceedings of the 3rd USENIX Virtual Machine Research and Technology Symposium*, 2004.

[81] Eric Youngdale. Kernel Korner: The ELF Object File Format by Dissection. *Linux Journal*, Vol. 1995, No. 13, 1995.

[82] Matt Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. http://msdn.microsoft.com/en-us/library/ms809762.aspx, 1994.

[83] Steven Mccanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *In Proceedings of the USENIX Winter 1993 Conference*, pp. 259–269, 1992.

[84] NT Kernel Resources: Windows Packet Filter Kit. http://www.ntkernel.com/w&p.php?id=7.

[85] GNU Binutils. http://www.gnu.org/software/binutils/.

[86] DUMPBIN Reference. http://msdn.microsoft.com/en-us/library/c1h23y6c.aspx.

[87] Intel Corporation. *Using the RDTSC Instruction for Performance Monitoring*, 1997.

[88] Crypto++ Library - a Free C++ Class Library of Cryptographic Schemes. http://www.cryptopp.com/.

[89] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, and M. Minier. Sosemanuk, a fast software oriented stream cipher. In *Proceedings SKEW - Symmetric Key Encryption Workshop Network of Excellence in Cryptology ECRYPT*, 2005.

[90] Paul Gray and Anthony Betz. Performance Evaluation of Copper-based Gigabit Ethernet Interface. In *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)*. 2002.

[91] Hyun-Wook Jin and Chuck Yoo. Impact of protocol overheads on network throughput over high-speed interconnects: measurement, analysis, and improvement. *The Journal of Supercomputing*, Vol. 41, No. 1, pp. 17–40, 2007.

[92] Usernet.com: User Information Center. http://www.usenet.com/.

[93] Microsoft Windows Server 2003 TCP/IP Implementation Details. Technical report, Microsoft Corporation, 2003.

[94] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 29–42. USENIX Association, 2001.

Appendix A

Syntax of the Configuration File

A.1 The Service Element

service element : *service/name service/library service/type* [*parameters*] [*parameter options*] [*insertion rule*]

service/name attribute : *network service name*

service/lib attribute : *network service library name(path)*

service/type attribute : *local* | *global*[*/required* | */optional*]

parameters : *service/parameter* [*parameters*]

service/parameter element : *parameter/name parameter/value*

parameter/name attribute : *parameter name*

parameter/value attribute : *parameter value*

parameter options : *service/parameter-option service/parameter-option*

service/parameter-option : *parameters*

insertion rule : *service/rule*

service/rule element : *rule/use rule/service rule/transport rule/port rule/type*

A.2 An Example of the Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration application="myserver">
  <services>
    <service name="firewall" lib="libfreena_fw.so" type="local">
      <parameter name="SQL-injection" value="ON"/>
      <parameter name="Abnormal" value="disconnect"/>
    </service>

    <service name="SSL" lib="libfreena_ssl.so" type="global">
      <parameter name="CA" value="rootcert.pem"
      <parameter name="PrivateKey" value="server.pem"/>
      <parameter-option>
        <parameter name="Version" value="SSLv3:TLS"/>
      </parameter-option>
      <parameter-option>
        <parameter name="Version" value="SSLv2"/>
      </parameter-block>
      <rule use="true" service="FTP" transport="TCP" port="8020-8021"
            type="*"/>
    </service>

    <service name="compression" lib="libcomp.so" type="global/optional">
      <param name="algorithm" value="Deflate"/>
    </service>

    <using-rules>
      <rule use="true" service="HTTP" transport="TCP" port="8080"
            type="server"/>
      <rule use="false" service="*" transport="*" port="*" type="*"/>
    </using-rules>
  </services>

  <protocol>
    <default name="TCP" lib="kernel"/>
    <option  name="SCTP" lib="libsctp.so">
      <parameter name="window-size" value="128K"/>
    </option>
  </protocol>
</configuration>
```

Figure A.1: A fully example of the configuration file

Appendix B

Implementation Notes

B.1 Recursive Socket Functions Calling Problem

In our approach, socket functions called by applications are replaced with functions of the control library by dyninst API. Dyninst API inserts jump instruction into process images at the beginning of socket functions' code. Therefore, when the interface library calls socket functions, control library's functions are eventually called again.

We solved this problem by using `syscall` system call on linux and customized *WinSock* library on windows. `syscall` can be used to call system calls by specifying function numbers. Customized *WinSock* library is almost the same with original *WinSock* except that function's names are slightly changed like `send` → `senX`. Since dyninst API identifies functions by their names, this approach is valid.

B.2 Supported Socket Functions

Common socket functions supported by FreeNA are listed in Tab. B.1. FreeNA user can modify the behavior of listed functions by preparing network service libraries.

Table B.1: Supported socket functions (AF_INET)

`socket`	`bind`	`connect`
`listen`	`accept`	`send`
`recv`	`sendto`	`recvfrom`
`close`	`shutdown`	`getpeername`
`getsockname`	`getsockopt`	`setsockopt`

B.3 Internal Socket Function Interposing

Figure B.1 shows an assembly-level call graph of `socket()` function on the Linux platform. This assembly code was acquired by dumping running process image after interposing specified socket functions by Dyninst API.

Application

```
0x08049090 <_Z14server_processbi+12>:   push   $0x0
0x08049092 <_Z14server_processbi+14>:   push   $0x1
0x08049094 <_Z14server_processbi+16>:   mov    0x8(%ebp),%al
0x08049097 <_Z14server_processbi+19>:   push   $0x2
0x08049099 <_Z14server_processbi+21>:   mov    0xc(%ebp),%esi
0x0804909c <_Z14server_processbi+24>:   mov    %al,-0x519(%ebp)
0x080490a2 <_Z14server_processbi+30>:   movl   $0xffffffff,-0x524(%ebp)
0x080490ac <_Z14server_processbi+40>:   call   0x8048ddc <socket@plt>
...
```

Procedure linkage table of socket() function

```
0x08048ddc <socket@plt+0>:  jmp    *0x804c160 (0x412b3f10)
0x08048de2 <socket@plt+6>:  push   $0xa0
0x08048de7 <socket@plt+11>: jmp    0x8048c8c
```

socket() function (overwritten by Dyninst)

```
0x412b3f10 <socket+0>:  jmp    0x400ad7f8 <DYNINSTstaticHeap_16M_anyHeap_1+4680>
0x412b3f15 <socket+5>:  int3
0x412b3f16 <socket+6>:  int3
0x412b3f17 <socket+7>:  int3
...
```

Dyninst trampoline for ctl_socket() function

```
0x400ad7f8 <DYNINSTstaticHeap_16M_anyHeap_1+4680>:  pusha
0x400ad7f9 <DYNINSTstaticHeap_16M_anyHeap_1+4681>:  push   $0x412b3f10
0x400ad7fe <DYNINSTstaticHeap_16M_anyHeap_1+4686>:  push   %ebp
0x400ad7ff <DYNINSTstaticHeap_16M_anyHeap_1+4687>:  mov    %esp,%ebp
0x400ad801 <DYNINSTstaticHeap_16M_anyHeap_1+4689>:  sub    $0x84,%esp
0x400ad807 <DYNINSTstaticHeap_16M_anyHeap_1+4695>:  leave
0x400ad808 <DYNINSTstaticHeap_16M_anyHeap_1+4696>:  pop    %eax
0x400ad809 <DYNINSTstaticHeap_16M_anyHeap_1+4697>:  popa
0x400ad80a <DYNINSTstaticHeap_16M_anyHeap_1+4698>:  push   $0x4135a154
0x400ad80f <DYNINSTstaticHeap_16M_anyHeap_1+4703>:  ret
...
```

ctl_socket() function of the control library

```
0x4135a154 <ctl_socket+0>:    push   %ebp
0x4135a155 <ctl_socket+1>:    mov    %esp,%ebp
0x4135a157 <ctl_socket+3>:    push   %esi
0x4135a158 <ctl_socket+4>:    push   %ebx
0x4135a159 <ctl_socket+5>:    sub    $0x20,%esp
...
0x4135a1b2 <ctl_socket+94>:   xchg   %ax,%ax
0x4135a1b4 <ctl_socket+96>:   sub    $0x4,%esp
0x4135a1b7 <ctl_socket+99>:   push   %edx
0x4135a1b8 <ctl_socket+100>:  push   %esi
0x4135a1b9 <ctl_socket+101>:  push   %ecx
0x4135a1ba <ctl_socket+102>:  call   *%eax  (function pointer)
...
```

service_socket() function of the networking service library

```
0x410b971c <service_socket+0>:  push   %ebp
0x410b971d <service_socket+1>:  mov    %esp,%ebp
0x410b971f <service_socket+3>:  push   %ebx
0x410b9720 <service_socket+4>:  sub    $0x8,%esp
...
0x410b9737 <service_socket+27>: pushl  0x10(%ebp)
0x410b973a <service_socket+30>: pushl  0xc(%ebp)
0x410b973d <service_socket+33>: pushl  0x8(%ebp)
0x410b9740 <service_socket+36>: call   *0x194(%eax) (function pointer)
...
```

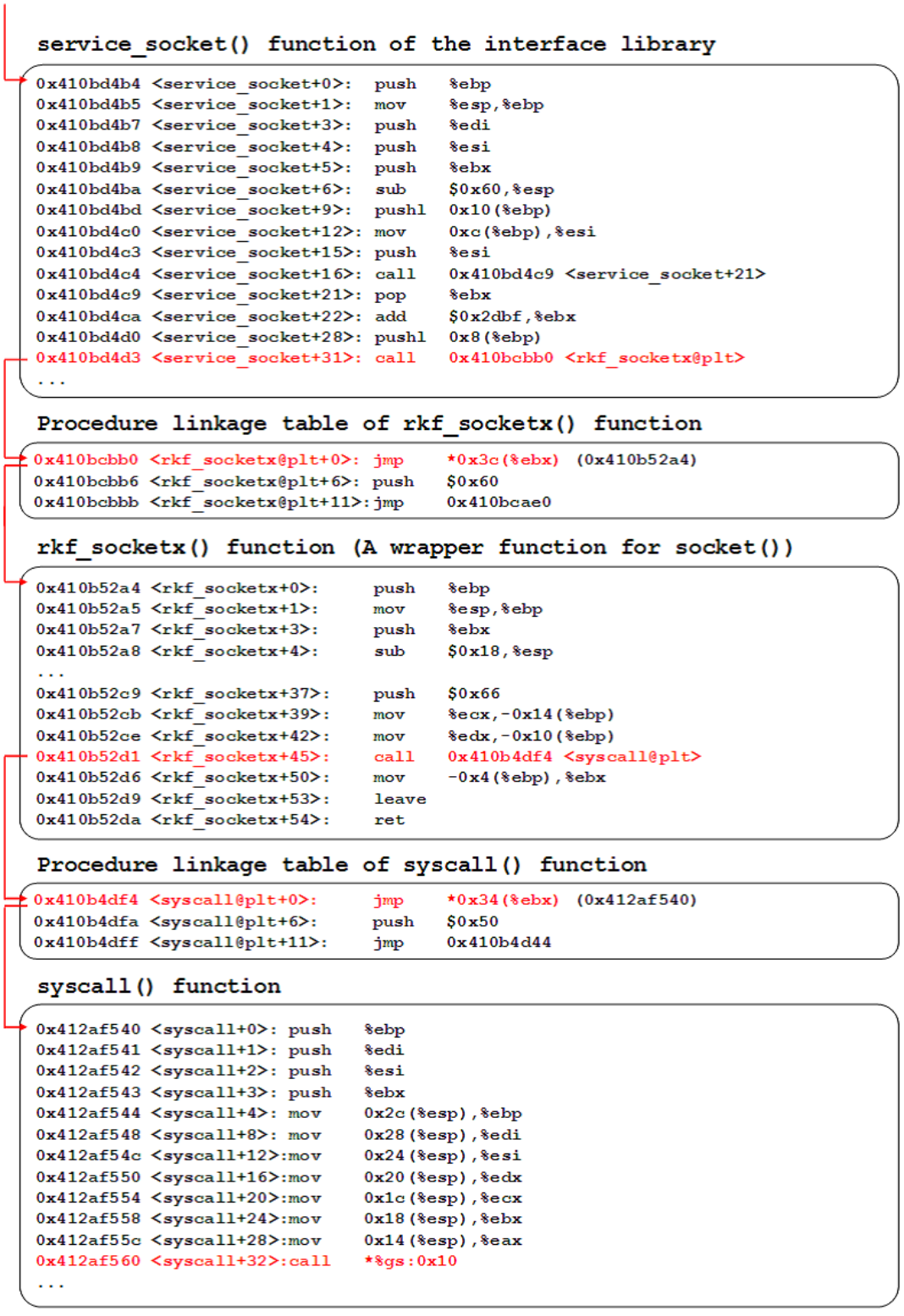

Figure B.1: Assembly-level socket function call graph

List of Publications

Journal papers

- Ryota Kawashima, Yusheng Ji, and Katsumi Maruyama, "FreeNA: A Multi-Platform Framework for Inserting Upper-Layer Network Services", IEICE Transactions on Information and Systems, vol.E92-D no.10, pp.1923-1933, Oct. 2009.
- 川島 龍太, 計　宇生, 丸山 勝巳, "トランスポート層プロトコルフリーな通信環境を透過的に実現するフレームワークの開発", 電子情報通信学会論文誌 B vol.J92-B no.4, pp.656-666, April 2009.

International Conferences

- Ryota Kawashima, Yusheng Ji, and Katsumi Maruyama, "Design and Implementation of Multi-Platform Infrastructure of Extensible Networking Functions", IEEE GLOBECOM 2008, New Orleans, LA, USA, 2008.

Refereed Domestic Conferences

- 川島 龍太, 計 宇生, 丸山 勝巳, "マルチプラットフォームにおけるネットワーク機能の透過的拡張のためのフレームワークの開発", FIT2008 情報科学技術フォーラム, 2008.

Domestic Conferences

- 川島 龍太, 計 宇生, 丸山 勝巳, "既存システムとの親和性を考慮した動的再構成可能な通信コンポーネントの提案", 情報処理学会第 7 回インターネットと運用技術研

究会, 2009.

- 川島 龍太, 計 宇生, 丸山 勝巳, ”透過的かつ適応的にトランスポート層プロトコルを変更するフレームワークの開発”, 情報処理学会第 3 回インターネットと運用技術研究会, 2008.

- 川島 龍太, 計 宇生, 丸山 勝巳, ”ネットワーク機能を透過的に拡張するミドルウェアシステムの提案”, 電子情報通信学会第 2 回ネットワークソフトウェア研究会, 2007.