ABSTRACT

WANG, ANJING. The SILO Architecture: Exploring Future Internet Design. (Under the direction of Dr. Rudra Dutta and Dr. George Rouskas.)

The architecture of the modern Internet encompasses a large number of principles, concepts and assumptions, that have evolved over several decades. In this dissertation, we argue that while the current architecture houses an effective design, it is not itself effective in enabling evolution.

To achieve architectural flexibility to enable evolution and to explore future Internet design, we introduce the SILO architecture, a meta-design framework within which the system design can change and evolve. We present some insights about architectural research that guided our work, as well as the goals we formulated for our architecture. We then describe that architecture itself, connecting it with relevant prior and current research work.

One of the great benefits of SILO is cross-service tuning optimization. We detail our approach to enable cross-service tuning, and then we present sample tuning algorithms, and experiential results.

In order to sustain the Internet evolving and to enable others to contribute, we introduce the architectural support for programming (services, tuning algorithms, and applications) in the SILO world. We also discuss an early case study on the usefulness of SILO in lowering the barriers to contribution and innovation in network protocols.

We validate the promise of enabling change by demonstrating how our recent study supports virtualization, as well as integration with other architectures.

Network virtualization has become a new trend in recent years and has been considered an important aspect of network architecture design. However, current understanding of network virtualization is still limited. We investigate the entire spectrum of network virtualization and propose a fundamental way to define network virtualization by using network resources. We also investigate its fundamental resources. In this context, we present a SILO architectural extension for network virtualization.

In an effort to promote SILO in deployment, we investigate the integration of the SILO architecture with several other architectures in GENI. IMF functions as SILO's measurement plan, while ORCA works as the resource control plan. We discuss our work and experimental results, and the flexibility of the SILO architecture in working with other architectures.

A discussion of various future works reveals our vision of the next steps to exploring the future Internet. SILO shows promise in various aspect, and will be the cornerstone of future research.

The SILO Architecture: Exploring Future Internet Design

by
Anjing Wang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2010

APPROVED BY:

Dr. George Rouskas Co-chair of Advisory Committee	Dr. Ilia Baldine
Dr. Vincent Freeh	Dr. Edward Gehringer

Dr. Rudra Dutta
Chair of Advisory Committee

DEDICATION

To my mom and dad, who put all their trust in me and supported my studies abroad;

To my wife, who gave up a good job and being close to her parents and friends in order to support me with all her heart;

To my little sister and her family, who have taken care of our parents while I am on my journey here; and

To the Pierces, for their love and care for our family.

BIOGRAPHY

Anjing Wang is a PhD candidate in the Department of Computer Science at North Carolina State University. He received his bachelor's and master's degrees from Department of Information and Communication, Xian Jiaotong University, China in 2000 and 2003, respectively. After working with Lenovo (Beijing), Nortel Networks (Beijing) and Nortel Networks (California), he joined NC State University under the instruction of Dr. Rudra Dutta (advisor) and Dr. George Rouskas (co-advisor). He is the author of several patents (registered in China) in the wireless LAN area. His current research focus upon networking architecture design.

Anjing served as an at-large officer of the Computer Science Graduate Student Association (CSC GSA) for two years, and he was also a representative of CSC GSA in the University GSA, which is a bridge between NCSU graduate students and CSC students. He was nominated as an Outstanding Teaching Assistant of the Graduate School and named Outstanding Teaching Assistant in the Computer Science Department of NCSU on Apr 24, 2009.

He will proudly join Ericsson in the next step of his career.

ACKNOWLEDGEMENTS

This work was funded by NSF Grants CNS-0626103, NeTS-FIND: The SILO Architecture for Services Integration, controL, and Optimization for the Future Internet, and by NSF GENI Project 1718, GENI IMF: Integrated Measurement Framework and Tools for Cross-Layer Experimentation.

I would like to express my gratitude to Dr. Rudra Dutta, Dr. George Rouskas, and Dr. Ilia Baldine, who have guided me throughout the research; to Mohan Iyer, who has worked with me regarding some architecture designs. Thanks to my wife, Hong, my parents and my sister. They have encouraged me throughout my graduate study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Current Internet Design

The Internet, conceived in the era of mainframe computers and 56Kbps links, has evolved into a complex world-wide system of importance equal to that of the power grid and the transportation infrastructure. The Internet's explosive growth has been mainly due to its innate ability to incorporate easily new link and node technologies, and to accommodate seamlessly novel protocols, applications, and edge devices. The Internet's successful evolution from a free tool for academic pursuits to a key component of the global information and communications infrastructure is a testament to the flexibility of its architecture and the fundamental principles underlying its design [1, 2]. The resulting combination of a simple, transparent network offering a basic communication service with end systems providing for a rich functionality, which lies at the foundation of the Internet architecture, has proven exceptionally adaptable to new and changing requirements.

While the network works well for common communication tasks, there have emerged communities of users with widely divergent needs for whom the current architecture is either inadequate or too complicated. For instance, the e-Science community, with its emphasis on high performance networking, currently supports the development of specialized protocols [3]. At the other end of the spectrum, the proliferation of network-enabled low-power mobile devices and sensors designed for simple, specific tasks represents an increasing need for a rudimentary communication service; for example, TinyOS did not implement the standard TCP/IP stack [4]. Continuing to ignore these divergent needs could result in balkanization of the Internet [5]. We believe that any new network architecture must possess a wide range of operating regimes and be able to deal gracefully with a broad spectrum of application requirements, network performance, and device capabilities, so as to ensure a unified, globally accessible Internet.

The suitability of protocol layering, as either an organizing or implementation principle

for future network architectures, is also being questioned [6]. Protocols typically incorporate significant functionalities, making them inflexible and difficult to evolve. New functionality is challenging to fit in the rigid structure of network stacks, resulting in the proliferation of 1/2 layer solutions (e.g., IPSec and MPLS). Furthermore, the lack of mechanisms for cross-layer interactions has led to frequent layer violations. Moreover, protocols were not designed to take advantage of emerging hardware architectures, such as the Cell [7], with one main and several synergistic processors in a single chip. A more modular design that makes it easier to selectively offload into hardware the most time consuming functions might lead to significant performance gains.

1.2 In Search of Future Internet Architecture

Despite the current success of the Internet, research communities have always actively sought to identify the future Internet architecture.

In 1972, Robert Metcalfe was famously able to capture the essence of networking with the phrase "Networking is inter-process communication." Nevertheless, describing the *architecture* that enables this communication to take place is by no means an easy task. The architecture of something as complex as the modern Internet encompasses a large number of principles, concepts and assumptions, which necessarily bear periodic re-visiting and re-evaluation in order to assess how well they have withstood the test of time. Such attempts have been made periodically in the past, but began truly coming into force in the early 2000s all around the world.

In the United States, there are quite a few initiatives in this direction. Since the 1990s, even before most people did not sense the incoming great success of the Internet, some researchers and Internet pioneers pointed out the new requirements of the Internet and proposed new architecture principles [8, 9]. From 2000, DARPA funded the NewArch project [10] to explore the new requirements brought about by the rapid growth of the Internet, and new architectural designs according to the changes. In 2004, SIGCOMM organized the Future Directions in Network Architecture workshop [11] to discuss what might be the next directions. FIND (Future Internet Design) [12] is an NSF-funded program starting founded in 2006 to encourage researchers to consider the Internet will look like in 15 years. What distinguishes this program is that FIND encourages researchers to design new networks from scratch without being constrained by the current Internet architecture. Forty-nine various projects were funded by NSF FIND, and the SILO project was one of them. NSF also sponsors GENI (the Global Environment for Network Innovations) [13], aiming to provide an infrastructure upon which researchers are able to deplore their own architectures and experiments. GENI is not intended to become the next Internet architecture itself, but it is recognized as serving the purpose of prompting

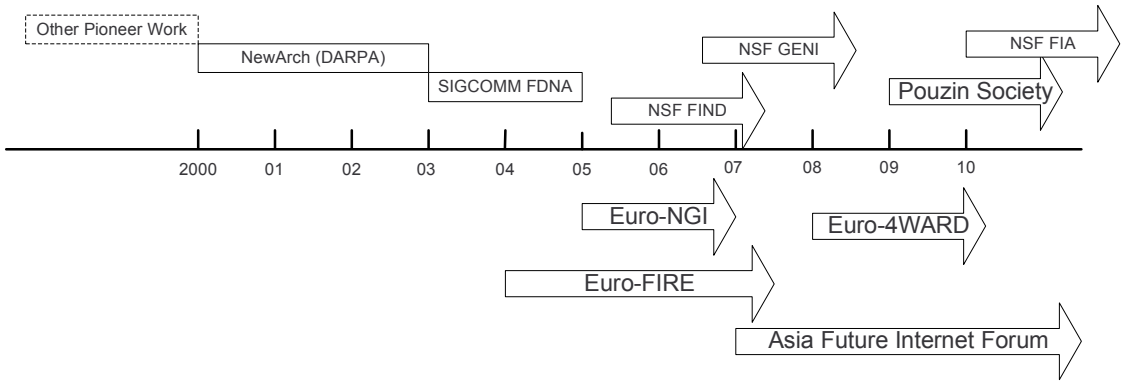

Figure 1.1: Timeline of Next Generation Internet Research Projects

and building the next generation Internet. NSF FIA (Future Internet Architectures) is the latest funding effort to encourage network architecture research.

In Europe, Euro-NGI has sponsored an annual conference regarding the next generation Internet since 2005. The Future Internet Research & Experimentation Initiative (FIRE) addresses the needs of the future Internet from various aspects. The 4WARD project [14] shows the ambition of these researchers to lead the change of the Internet. It is a three-year long project that began in 2008 and addresses two aspects of network design. One is from the perspective of architecture, while the other is from the view of in-network management. The task focuses on WP2 New Architecture Concepts and Principles. The European Future Internet Portal [15] collects various efforts made by researchers in the EU about the next generation Internet. A recently published book, "*Towards the Future Internet - Emerging Trends from European Research*" [16], summarizes ongoing projects and efforts by the EU. Overall, research activities about the next generation Internet in the EU are very active.

The Asia Future Internet Forum (AsiaFI) was founded to coordinate R&D on the future Internet among Asia counties. China launched the China Next Generation Internet (CNGI) and 973 New Generation Internet Architecture in 2003. Japan started AKARI (Design a New Generation Internet) in 2006. South Korea has made efforts through the Future Internet Forum (FIF) since 2006.

Figure 1.1 summarizes the timeline of Next Generation Internet research endeavors since 2000.

1.3 Related Work on Composable Architecture

1.3.1 Function-based Communication Subsystem

The Function-based Communication Subsystem[17] was published in 1993. As the insight of layering architecutre is not enough, the following concepts are proposed:

(1) application interfered network;

(2) fine granularity; and

(3) configurable information units.

Figure 1.2 shows the architecture of F-CSS. Applications are able to open multiple sessions to communicate and interact with the Session Manager. The Session Manager also oversees Protocol Machines in its session. F-CSS primarily deals with the network stack above multiplex/demultiplex.

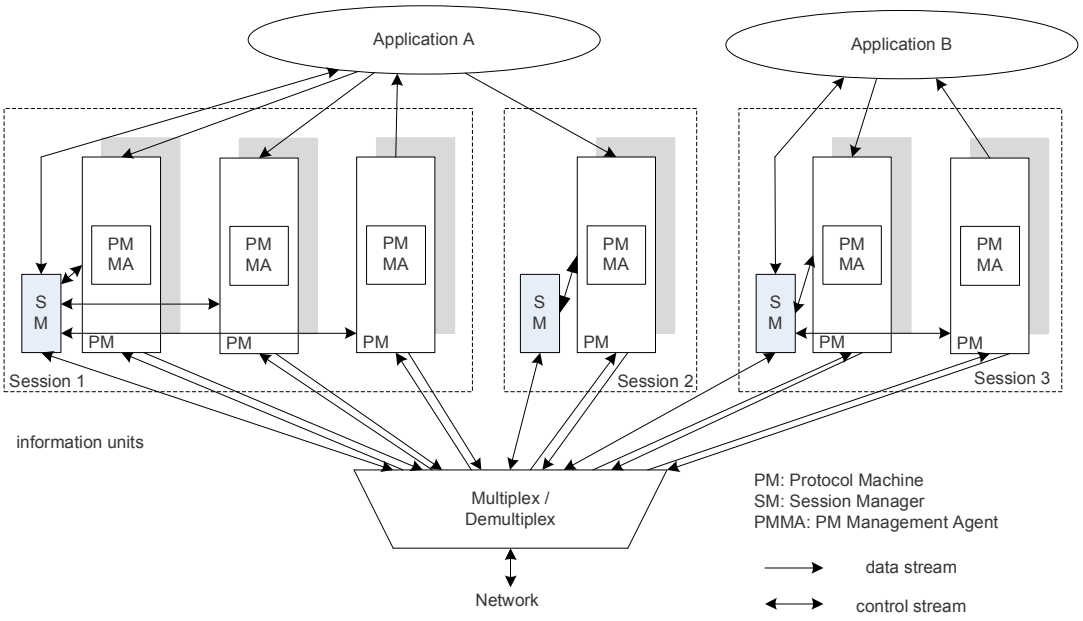

Figure 1.2: F-CSS Architecture (adapted from [17])

1.3.2 *X*-kernel

X-kernel [18] is the pioneer in investigating flexible frameworks for implementing network protocols. The implementation of *x*-kernel started in the early 1990s, but was out of maintenance

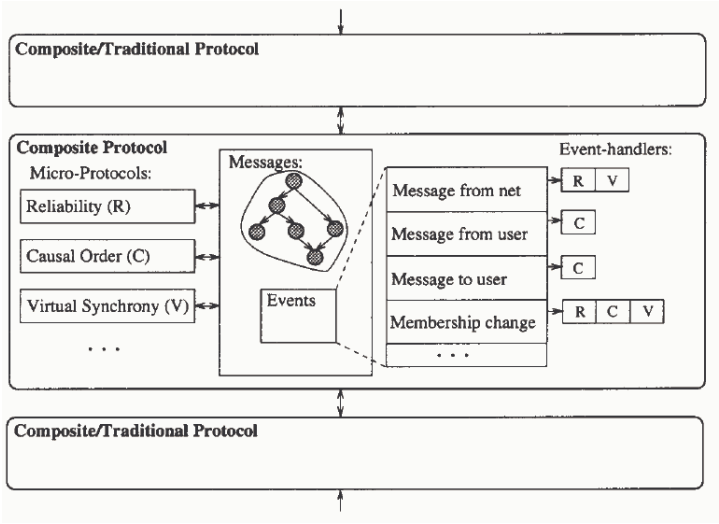

Figure 1.3: Composite Protocol in Coyote (Reproduced from [21])

by 1997. The main goal of x-kernel is to show that a modular, highly structured way is general enough for implementing most of the network protocols.

In the x-kernel world, a protocol is an object. It uses "service interfaces" to interact with protocols in the same host, and uses "peer-to-peer interfaces" for protocols in the remote hosts. X-kernel only defines the syntax for interfaces of protocols, not the semantics of protocols.

A dynamic network architecture [19] is a network architecture based on x-kernel.

1.3.3 Coyote: Fine-Grain Configurable Services

Coyote [20, 21] extends x-kernel coarse protocols and introduces composite protocols and microprotocols. Figure 1.3 shows a sample of the composite protocol.

A composite protocol is like a traditional protocol, such as TCP or OSPF, which can be used to form a network subsystem. Similarly to x-kernel, these composite protocols have uniform interfaces, such as pop() and push(). A list of coyote service/composite protocols can be used to construct a network stack.

A suite of microprotocols creates a composite protocol. Microprotocols can share variables, such as event handlers and so on. Microprotocols can be statically linked together to form composite protocols.

The prototype of Coyote is based on x-kernel.

1.3.4 Role-Based Architecture

With the idea of fine-grained services or so-called microprotocols, researchers realized that there should be control components to manage these services. Role-Based Architecture (RBA) [6, 10] represents a non-layered approach to designing network protocols and organizing communication in functional units, referred to as "roles." Roles are not hierarchically organized; as a result, the metadata in the packet header corresponding to different roles form a "heap," not a "stack," and may be accessed and modified in any order. RBA allows re-modularization of protocols, such as IP and TCP, to smaller units.

1.3.5 A Recursive Network Architecture

A Recursive Network Architecture (RNA) [22] is also in the FIND program. It challenges the current Internet architecture in terms of the static nature of conventional stacks, recapitulates services in different layers, and adds a variety of layers by visualization. It further addresses three observations. Firstly, services should not be specific to a particular layer and can be deployed at a number of layers. Secondly, recapitulating/reimplementing services should be avoided. Finally, a generic way should be proposed to coordinate the first two observations. As a result, a metaprotocol was proposed, which can be recursively used to construct a networking stack. Although the metaprotocol is an abstraction of a generic protocol, it is not just a template. It is an instance of a protocol, which is configurable to suit different layers in networking stacks.

1.3.6 Tng: Transport Next-Generation

Unlike other related projects described above, Tng [23] only focus on decomposing the transport layer into several transport functions, such as flow regulation and endpoint. It also introduces a cross-layer negotiation protocol [24] acting as a negotiator to set up a network stack.

1.4 Our Exploration

This thesis presents a new architecture as part of our efforts to explore the future Internet, the SILO architecture. It is an joint effort pursued at North Carolina State University and the Renaissance Computing Institute, and is made possible through NSF grants CNS-0626103 and CNS-0732330.

The rest of this thesis is organized as follows. In Chapter 2, we present our design philosophy and principles, the SILO architecture, and SILO software prototype framework. Next, in Chapter 4, we explore architectural support for cross-service tuning. In Chapter 5, we present architectural support for programming services, tuning algorithms, and applications.

In Chapter 6, we investigate fundamental concepts of network virtualization and present SILO's corresponding extension in Chapter 7. Chapter 8 details integration of the SILO architecture in GENI, the IMF project. Various considerations for future work are presented in Chapter 9, and the thesis is concluded in Chapter **??**.

Chapter 2

The SILO Architecture

In this chapter, we explore the design and implementation of SILO, a new clean-slate network architecture that represents a departure from current philosophy and practice. We outline an architecture consisting of:

(1) building blocks of fine-grain functionality;

(2) explicit support for combining elemental blocks to accomplish highly configurable complex communication tasks; and

(3) control elements to facilitate the architecture.

With a holistic view of network design, SILO allows applications to work synergistically with the network architecture and physical layers to select the most appropriate functional blocks and tune their behaviors, so as to meet the applications' needs within resource availability constraints.

This chapter is organized as follows. In Section 2.1, we elaborate on our design philosophy and principles. In Section 2.2, we present the basic design and components of SILO architecture. We conclude the chapter in Section 2.3 by comparing SILO to related architecture designs.

2.1 Design Philosophy and Principles

SILO architecture design is echoed by our design design philosophy and principles. We will detail our design philosophy and key design principles in this section and explain why we design our architecture the way it is.

2.1.1 SILO Philosophy: Designing for Change

The "To Change or Not to Change" Tussle

The degree to which the Internet continues to permeate modern life, with hundreds of new uses and applications adapted to various networking technologies (from optical, to mobile wireless, to satellite), raises concerns about the longevity of the Internet architecture. The original simple file transfer protocols and UUNET gave way to e-mail and WWW, which by now are becoming eclipsed by streaming media, compute grids and clouds, instant messaging and peer-to-peer applications. Every step in this evolution raises the prospect of re-evaluation of the fundamental principles and assumptions underlying the Internet architecture. So far, this architecture has managed to survive and adapt to the changing requirements and technologies, while providing immense opportunities for innovation and growth. On the one hand, such adaptability seems to confirm that some of the original principles have truly been prescient to allow the architecture to survive for over 30 years. On the other hand, it raises the question of whether the survival of the architecture is in fact ensured by the reluctance to question those principles, cemented by shoehorning novel applications and technologies into the existing architecture without considering its suitability.

Such a contradiction will not be easily resolved, nor should it be. A dramatic shift to a new architecture should only be possible for the most compelling of reasons, and so, the existence of this contradiction creates the ultimate "tussle" [25] for the networking researcher community. This tussle pits the investment in time, technologies and capital made in the existing architecture against adapting the new architectures. Adapting new architectures open up possibilities of allowing for the creation of novel and improved services over the Internet, as well as exploring new areas of research and discovery. It also allows us to continually refine the definition of the Internet architecture and to separate and re-examine various aspects of it. A sampling of the projects funded through the NSF FIND program, targeted at re-examining the architecture of the Internet, illustrates the point; there are projects concerned with naming [26, 27], routing [28, 29], and protocol architectures [30], which examine these and other aspects from perspectives of security, management [31], environmental impact [32] and economics [27]. Another dimension is presented by the range of technologies allowing devices to communicate: wireless, cellular, optical [33] and adaptations of the Internet architecture to such devices.

This diversity of points of view makes it difficult to see clearly the fundamental elements of the architecture and their influence over each other. Most importantly for the researchers interested in architecture, this makes it nearly impossible to answer concisely the question of what the Internet architecture actually is, or even what concerns are encompassed by the term "Internet architecture." What things should be considered part of the architecture of a complex

system, and what should be considered specific design decisions that are comparatively more mutable? This further fuels the "to change or not to change" tussle we mentioned to above.

Designing for Change

One way to make progress in the tussle is to create modifications in the current architecture that enable new functionality or services not possible today, while limiting the impact on the rest of the architecture in essence, evolving the architecture while preserving backward compatibility [34]. This approach has the additional merit of paying heed to the concerns expressed by some in the research community regarding the potential of clean-slate approaches to be far divorced from reality, with no reasonable chance of translating to deployment [34]. Such concerns have been epitomized by the phrase "Clean-slate is not blue-sky."

The SILO project was, in a way, founded by following this approach. We did not attempt to rethink the Internet as a whole. Instead, we identified one particular aspect of the Internet architecture that, in our opinion, has created a significant barrier to its future development. We proposed a way to modify this aspect that least significantly impacts the rest of the architecture and demonstrated the use of this new architecture via a prototype implementation and case studies.

Somewhat to our surprise, however, what emerged from our research was a new understanding regarding the problem at hand. Specifically, we came to recognize that the important problem is *not* to obtain a particular design or arrangement of specific features, but rather, to obtain a *meta-design* that explicitly allows for future change. With a system like the Internet, the goal is not to design the "next" system, or even the "next best" system, but rather a system that can sustain continuing change and innovation.

This principle, which we call *designing for change*, became fundamental to our project. In the process, we have come to develop our own answer to the question of what architecture actually is: *it is precisely the characteristics of the system that do not change themselves, but provide a framework within which the system design can change and evolve.* The current architecture features an effective design, but is not itself effective in enabling evolution. Our challenge has been to articulate the necessary minimum characteristics of an architecture that will be successful in doing so.

2.1.2 Three Principles

As a starting point, we adopted the view that the layering of protocol modules within a data flow is a desirable feature that has withstood the test of time, as it makes data encapsulation easy and simplifies buffer management. The layer *boundaries*, on the other hand, do not have to be in specific places; to our minds, this causes the entrenchment of existing protocols and is

one of the causes of the identified ossification of the Internet architecture. Based on this initial assumption, the desirable characteristics of a new architecture to *generalize protocol layering* started to emerge:

(1) each data flow should have its own arrangement of layered modules, such that the application or the system could create such arrangements based on application needs and underlying physical layer attributes;

(2) the constituent modules should be small and reusable to assist in the evolution by providing ready-made partial solutions; and

(3) the modules should be able to communicate with each other over a well-defined set of interfaces.

These three principles became the basis of the SILO architecture, which is shown in Figure 2.2(b). From Principle-1, we derive each individual layered arrangement serving a single data flow as a *silo*. We refer to individual layers within a silo as *services/methods*, which comes from Principle-2. *Crossing-Layering Tuning* and *Knobs* are from Principle-3. A more detailed explanation is provided below.

Support for Service Deployment and Composition

As the system evolves, new reusable modules (services and methods) may be implemented and deployed to fulfill the changing requirements of various applications, while allowing the reuse of existing ones. One may think of a downloadable driver/plug-in model as being appropriate in this context – new services and methods may be added to the system via one or more trusted remote repositories.

As not all modules can be assumed to be able to co-exist with each other in the same silo, it is necessary to monitor module compatibility. We refer to these as *composability constraints*. These constraints may be specified by the creators of the modules when the modules are made available, or they may be automatically deduced based on the description of module functionality. We envision that knowledge distilled from deployment experience of network operators, collectively, can also be stored here. The number of such constraints can be expected to be large and to grow with time. This highlighted the need for automated silo composition, which can be accomplished by one or more algorithms based on the application specification. This automated construction of silos became a crucial part of the architecture.

Support for Cross-Layer Interactions and Control

From the perspective of cross-layer interactions, it also became desirable to not simply allow modules to communicate with each other outside the data flow, but to allow for an external

entity to access module states for the purposes of optimizing the behavior of individual silos and/or the system as a whole. We refer to this function as *cross-service tuning*, and it is accomplished by querying individual modules via *gauges* and modifying their state via *knobs*. Both gauges and knobs must be well defined and exposed as part of the module interface. The important aspect of this approach is that the optimization algorithm can be pluggable, just like the modules within a silo, allowing for easy re-targeting of optimization objectives by a substitution of the optimization algorithm. This addresses the previously identified deficiency of the current architecture, where policies and methods in protocol implementations are frequently mixed together, not allowing for the evolution of one without affecting the other.

Services vs. Methods

Borrowing from object-oriented programming concepts, what we call services are generic functions (such as *encryption*, *header checksum*, or *flow control*), while methods are specific implementations of services. Thus, in some sense, methods are polymorphically associated with services. This relationship allows for the aggregation of some composability constraints based on generic service definitions, which necessarily propagate the methods implementing this service, thus making the job of the developer, as well as of the composition algorithm, substantially easier.

2.1.3 Support for Evolution and Innovation

Let us now address the issue of why this architecture is better suited for evolution than the current one. As mentioned in the previous section, our mantra for this project has been "design for change," and we believe we have succeeded in accomplishing this goal. The architecture we have described does not mandate that any specific services be defined or methods implemented. It does not dictate that the services be arranged in a specific fashion and leaves a great deal of freedom to the implementors of services and methods. What it does define is a generic structure in which these services can coexist to help applications fulfill their communications needs, which can vary depending on the type of application, the system it is running on, and the underlying networking technologies available to it. Thus, as the application needs to evolve along with the networking technologies, new communications paradigms can be implemented by adding new modules into the system. At the same time, all previously developed modules remain available, ensuring a smooth evolution.

The described architecture is a *meta-design* that allows its elements (the services and methods, the composition and tuning algorithms) to evolve independently, as applications require change and networking technologies evolve. Where, in the current architecture, the IP protocol forms the narrow waist of the hourglass (i.e., the fundamental invariant), in the SILO archi-

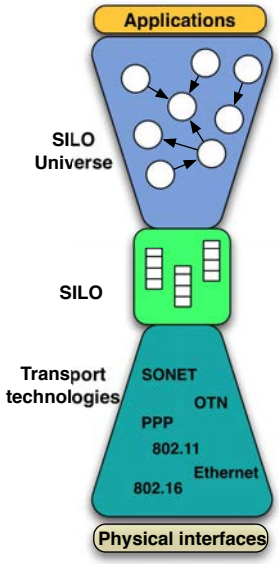

Figure 2.1: The SILO Hourglass

tecture, the convergence point is the conformance to the meta-design, *not* a protocol (which is part of the design itself). Rather than a protocol upon and under which all else must be built, SILO offers the silo abstraction as an invariant, the narrow waist in the hourglass of this meta-design (Figure 2.1).

2.2 Basic Architecture

Figure 2.2 shows the difference between the SILO approach and the traditional networking stack. The traditional stack is shown in Figure 2.2 (a). There are multiple applications to which the transport layer provides sockets, but only a single instance of all other layers. This makes it impossible to provide customized service to different applications. For example, if we want to use reliable transport, we have to use the TCP/IP stack. It is quite hard to extract this functionality gracefully, while eliminating some other TCP functionality for some application that does not need it. Another downside of this traditional stack is that we cannot easily perform cross-layer tuning to optimize the stack; we have to write a custom version of layers for that purpose.

However, the SILO architecture incorporates these two requirements, as Figure 2.2 (b) demonstrates. The networking communication entity is dynamically composed of fine-grained functionalities, which we call ***services***. The realization of services are called ***methods***. $m_{1,1}$, $m_{1,2}$, etc. in Figure 2.2 (b) refer to these methods. Every such service provides explicit interfaces

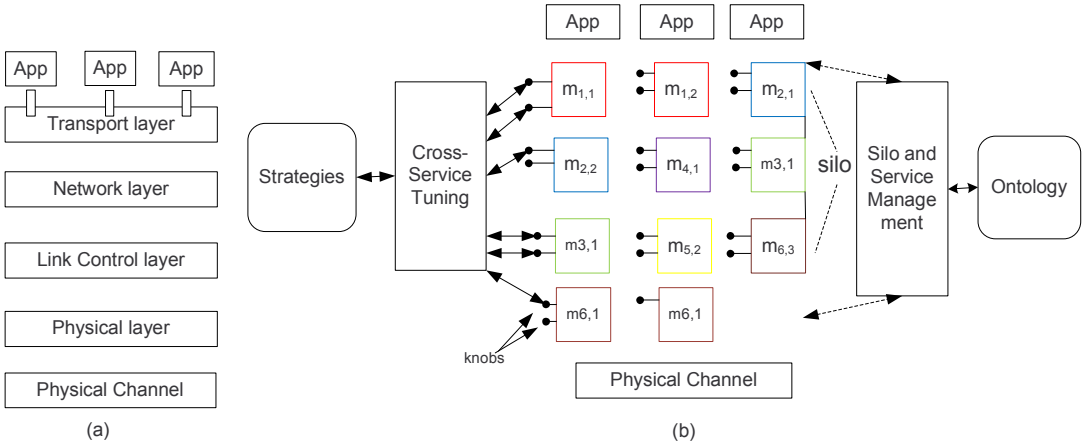

Figure 2.2: TCP/IP Stack vs. SILO Architecture

(***knobs***), which can be used by ***Cross-Service Tuning*** for performance optimization according to ***Strategies***. A stack of methods from an application to a physical channel, such as ($m_{1,1}$, $m_{2,2}$, $m_{3,1}$, $m_{6,1}$), is called a ***silo***[1]. ***Silo and Service Management*** composes silos with constraints from ***Ontology***. We explain these architectural components in greater detail next in the following several subsections.

2.2.1 Services

The fundamental building blocks in the SILO architecture are *services*. A service is a well-defined and self-contained function performed on application data, and it is relevant to a specific communication task. "In-order packet delivery," "end-to-end flow control," "packet fragmentation," "compression," "encryption," and "multi-rate RF PHY" are all examples of services in this context. Each service addresses a separate, atomic function; hence, the architecture provides more flexibility and much finer granularity than current protocols that typically embed complex functionality.

At the core of the architecture is the mechanism through which services interact in order to accomplish complex communication tasks. Our approach represents a middle ground between the strict protocol stack imposed by current architectures and the "heap" approach advocated by the RBA [6]. Specifically, we allow any set of services to be selected dynamically for a particular task, but the order in which these services are applied is not tied to the "layer" in which the service belongs, but rather to a set of well-defined precedence constraints; for instance, when the application requires both a "compression" and an "encryption" service, the

[1]We use the term "silo" (all lower-case) to refer to a stack of services; we reserve the term "SILO" (all upper-case) for the architecture itself.

only meaningful interaction is when compression is applied before encryption. In general, the precedence constraints impose a partial ordering among services. Once selected, however, the subset of services is arranged in a specific order, derived from the partial ordering and other rules, and this binding remains in effect for the duration of the associated communication task (typically, the lifetime of a connection).

A service is fully defined by describing: (1) the function it performs, (2) the interfaces it presents to other services, (3) any properties of the service that affect its relation with other services (e.g., as required to establish a partial ordering), and (4) its control parameters, which we also refer to as *knobs* (defined in 2.2.4) and their actions and constraints.

2.2.2 Methods

We distinguish between a service and its realization. A *method* is a realization of a service that uses a specific mechanism to carry out the functionality associated with the service. For instance, "re-sequencing" is one method for implementing the "in-order packet delivery" service, "window-based flow control" is a method for the "end-to-end flow control" service, and "802.11a OFDM PHY" is one method for the "multi-rate RF PHY" service. A method implementing a service must implement the service-specified interfaces, as well as any service-specific knobs; in other words, service-specific interfaces and knobs are polymorphic to all methods implementing a given service. A method may also implement method-specific knobs, i.e., control parameters unique to this implementation of a service; for instance, "length of Reed-Solomon FEC" is a knob specific to the "Reed-Solomon FEC" method implementing the "error-free delivery" service. These knobs are adjusted by *Cross-Service Tuning* to refine the method behavior and to optimize it for a specific environment.

A method is fully defined by describing (1) the service it implements, (2) the specific algorithm/mechanism it uses to implement the service, and (3) the optional method-specific control parameters, and their actions and constraints. We emphasize that the architecture defines services and their interfaces, but it does not define the methods that implement them; therefore, it is possible that several alternative methods for a given service co-exist within the network.

2.2.3 Silo

We refer to an ordered subset of methods, each method implementing a different service, as a *silo*. One can think of a silo as a vertical stack of methods; conceptually, applications reside at the top of the stack, and network interfaces reside at the bottom. A silo performs a set of transformations on data from the application to the network, or vice versa, so that the delivery of data from an application to its peer is consistent with the application's requirements. Each data transformation corresponds to a method in the silo and may include the generation (or

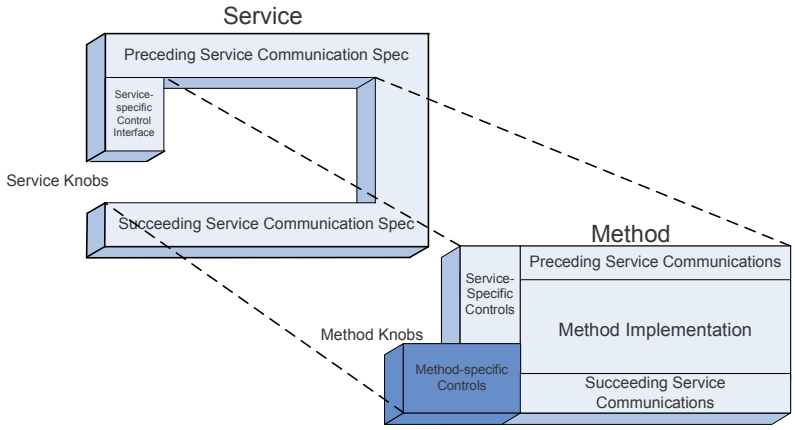

Figure 2.3: Service, Method and Knob

processing) of metadata to be included (respectively, present) in the packet. A silo possesses a state that is a union of all constituent method states, as well as any shared state resulting from cross-method interactions. A silo structure and all related state information are associated with a specific traffic stream (equivalently, a connection or flow), and they persist for the duration of the connection. One important aspect of silos is that they can be optimized for each traffic stream or for overall traffic stream, depending on the *Strategies* used by *Cross-Service Tuning*.

2.2.4 Knobs

The knobs include read/write knobs and read-only knobs. Alternatively, we call read/write knobs *knobs*, and read-only knobs *gauges*. The read/write knobs are adjustable parameters specific to the function performed by a service or a method, with a specified range of values or several enumerated values and a pre-defined relationship between these values and the perceived performance of the service or method. For instance, "compression factor" is a read/write knob for the "compression" service. The read-only knobs are values or states of a service or a method. For example, "throughput" is a read-only knob for the "performance monitor" service.

Figure 2.3 shows the relation between service, method and knob.

2.2.5 SILO Control

SILO control functionalities to support this architecture can be broken down into two classes, which are explained below.

Cross-Service Tuning

Cross-Service Tuning is an architectural element that has the responsibility of adjusting all the service- and method-specific knobs and facilitating cross-service interactions in order to obtain the desired performance. Cross-Service Tuning makes use of *Strategies* to select appropriate tuning algorithms. The details of Cross-Service Tuning will be discussed in Chapter 4.

Silo and Service Management

Silo and Service Management manages the different elements of SILO architecture and has the responsibility of composing a silo for an application stream (or selecting an appropriate commonly-used silo). The services constraints are stored in *Ontology*, which is used by SILO management to determine the subset of services it contains, their order in the stack, and the method implementing each service. The objective is to dynamically custom-build a silo for each new connection. To this end, the Services and SILO Management takes into account the application's QoS requirements, current network resource availability and other conditions, the precedence constraints among services, and any policy in effect at the time.

2.3 Comparisons with Similar Architectures

So far, we have presented all basic concepts of the SILO architecture. We summarize this chapter by comparing features of the SILO architecture with other composable architectures discussed in Section 1.3. Table 2.1 shows these feature comparisons.

In this chapter, we present the SILO philosophy, the design principles, the architecture itself, and the prototype framework. In the next two chapters, we focus on two aspects enabled by the SILO architecture: cross-service tuning and programming support.

Table 2.1: SILO Architecture vs. Other Composable Architectures

	Granularity	**Minim Unit**	**Tuning**	**Composibility**	**APP-customized network**
SILO	fine	service	cross-service tuning	ontology-aided composition	APP customized ontology or strategy
F-CSS	fine	protocol function	only implied	predefined composition	APP-interfered network stack
x-kernel	coarse	protocol	N/A	static composition	N/A
Coyote	fine	composite protocol and micro-protocol	briefly mentioned, not architecturally enabled	composed of constraints, not architecturally enabled	N/A
RBA	relatively small	role	N/A	small roles compose a complex role	N/A
RNA	N/A	metaprotocol	metaprotocol can be configured to different protocol	recursively compose of metaprotocol	N/A

Chapter 3

The SILO Software Prototype

We have developed a working prototype implementation that serves as a proof-of-concept demonstration of the feasibility of the SILO architecture, though we believe there are various ways to implement SILO architecture, and there might be better ways. Our software prototype is detailed in this section, and we also explain our choices and reasonings.

Our prototype software architecture (we also call it the SILO framework[1]) is shown in Figure 3.1. The SILO framework consists of the following major components:

- the SILO-enabled application (APP)
- the SILO API
- the SILO management agent (SMA)
- the SILO tuning agent (STA)
- the SILO construction agent (SCA)
- the SILO ontology of services and methods
- the universe of services storage (USS)

All the components shown in Figure 3.1 have been implemented. Firstly, we describe every component in Section 3.1. Then, we show how a silo is set up and interactions between components in detail in Section 3.3. Finally, we lay out implementation choices in Section 3.4 and software releases in 3.5.

[1]We deliberately use *SILO framework* to refer to our particular prototype, i.e., SILO prototype architecture, while reserving *SILO architecture* for the meta-design presented in Section 2.2.

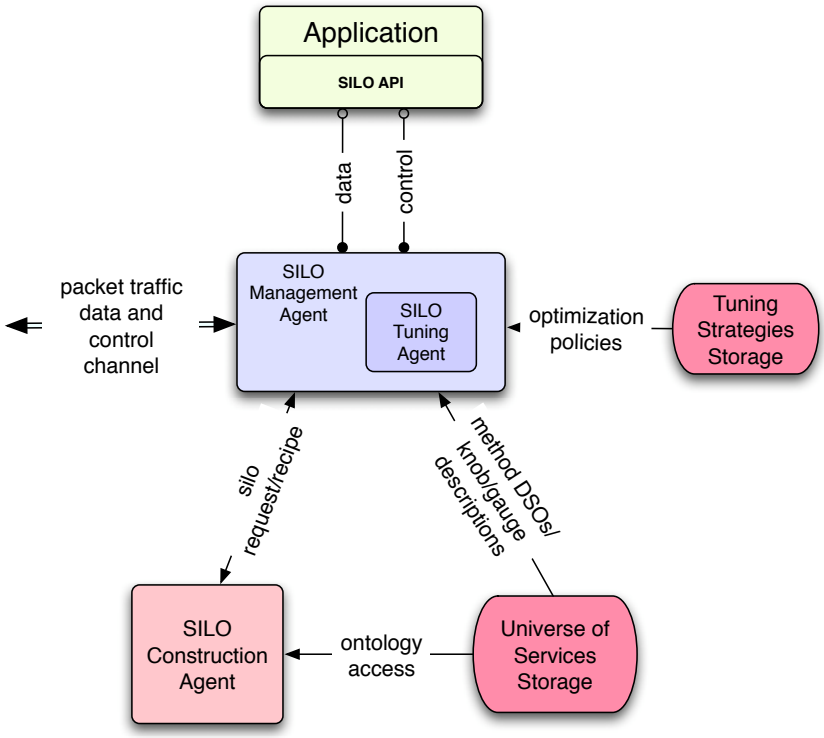

Figure 3.1: SILO Prototype Architecture

3.1 Prototype Component Description

3.1.1 The SILO-Enabled Application

The SILO-enabled Application (APP) is any application that performs networking communication through the SILO framework. This means the APP is linked against the SILO Application API and communicates with other SILO components through the SILO API. The APP could be a purpose-built application utilizing the SILO framework, or, alternatively, an existing networked application whose socket-based TCP/IP interface has been replaced with the SILO API or redirected to a SILO Application Gateway (details in Section 5.3.1).

3.1.2 The SILO API

The SILO API consists of C++ header files and library code. Two types of API and libraries are implemented:

- Application API - for creating SILO-enabled applications
- Internal API - library code common to the individual SILO components (e.g., to facilitate

inter-component communications)

Application API is exposed to applications. An application may call this set of APIs to request a silo, release a silo, send or receive data from a silo. Application API also allows applications to add application-specified constraints or ontology as part of the request.

Internal API is transparent to applications. The requests formed by applications are transferred to SMA, then forwarded to SCA by SMA. We do not allow applications talking with SCA directly, simply because we would like to maintain a single channel between applications and the management components.

3.1.3 The SILO Management Agent

The SILO Management Agent (SMA) is the most important part of the prototype. It is mainly responsible for (a) constructing a silo for a specific application based on a recipe created by the SCA by loading services from the Universe of Services Storage into memories, (b) handling requests from SILO-enabled applications and returning silo handlers, and (c) maintaining the silo state during the communications session.

As stated before, the SMA uses an event-driven programming model. We chose the live 555 library [35] because it provides the friendly unix socket interfaces we need.

The construction of a silo involves instantiating the silo state, loading the necessary method code from the Universe of Services Storage and starting any required execution context.

3.1.4 The SILO Tuning Agent

In SILO release 0.1, the tuning agent was a separate process and tuning algorithms were statically compiled together. Since the release of 0.2, the tuning agent has been integrated with the SMA and become part of it. The tuning algorithm is complied as a dynamically linked object and can be loaded by the tuning agent when necessary.

The change is based on the following reason. Although we can view the SILO Tuning Agent (STA) as a separate component from the SILO Management Agent conceptually, we do not need to develop another scaffold for the STA because the SMA already provides the perfect container to incorporate the STA functionalities. The STA is plugged into the SMA by setting scheduled events in the SMA.

The STA manipulates the knobs within individual silos in order to optimize its behavior according to a specific optimization goal. Optimization can be based on either individual or collective behavior of silos within a single node or among many nodes.

The significance of the STA will be discussed in great detail in Chapter 4.

3.1.5 The SILO Construction Agent

The SILO Construction Agent (SCA) is also a major component of the architecture, and its responsibility is to give out a reasonable silo recipe based on the application's request and composability constraints in Ontology.

The SCA only gives the recipe of a silo, represented in XML format. As mentioned before, the real construction work (loading services into the memory) is done by SMA. One might argue that some term like SILO Recipe Generator is a more proper name for this agent. However, we inherit this name from our initial architecture design as it was.

It could also be argued that we implement the SCA as a separate process, while integrating the STA as part of the SMA.

In SILO release 0.1, the SCA utilizes SILO Ontology, which is written in RDF. In order to access RDF by existing interfaces, we decided to use Jena, a set of Java API. So the first prototype was written in Java. We use TCP sockets to communicate between this Java-written SCA and the rest of the C++ written SILO components. That was the historical reason why the SCA and the SMA became separate processes.

In SILO release 0.2, we rewrote the SCA in Python, because Jena is a heavy library and we want to keep our release as light as possible. Besides, the original construction algorithm had various bugs, and it took less effort to rewrite the whole SCA than to debug the original algorithm. In the meantime, the rdflib python library provides friendlier and simpler interfaces to access RDF.

3.1.6 The SILO Ontology

One of the important challenges we encountered when addressing the problem of dynamically composable silos was related to the problem of representing the relationships (composability constraints) between the various services and modules. Essentially, this is a problem of knowledge representation. These composability constraints take the form of statements similar to "Service A requires Service B" or "Service A cannot coexist with Service B," which can be modulated by additional specifications such as 'above' or 'below' or 'immediately above' or 'immediately below.' Additionally, we needed to deal with the problem of specifying application preferences or requests for silos, which can be described as *application-specific* composability constraints. To address this problem, we turned to *ontologies*, specifically, ontology tools developed by the semantic Web community.

We adopted the RDF (Resource Description Framework)[36] as the basis for ontology representation in the SILO framework and use Protégé [37] as our ontology editor. RDF is a metadata model and uses triples to describe resources. RDF Schema (RDFS)[38] is a RDF vocabulary description language to semantically extend RDF and allows basic inferences.

Relying on RDF-XML syntax, we were able to create a schema defining various possible relationships between the elements of the SILO architecture, namely, services and methods. These relationships include the aforementioned composability constraints, which can be combined into complex expressions using conjunction, disjunction and negation. Using this schema, we have defined a sample ontology for the services we implemented in the current prototype. The application constraints/requests are expressed using the same schema. This uniform approach to describing both application requests as well as the SILO ontology is advantageous in that a request, issued by the application and expressed in RDF-XML, can be merged into the SILO ontology to create a new ontology with two sets of constraints – the original SILO constraints and those expressed by the application, on which the composition algorithm then operates. Using existing semantic Web tools, we have implemented several composition algorithms [39] that operate on these ontologies and create silo recipes, from which silos can be instantiated.

Our RDF schema also allows us to express other knowledge, such as the functions of services (an example of a service function could be "congestion control" or "error correction" or "reliable delivery"), as well as their data effects (examples include cloning of a buffer, splitting or combining of buffers, transformation, and finally null, which implies no data effect). These are intended to aid composition algorithms in deciding the set of services to be included in a silo, when an application is unable to provide precise specifications in the request. Using this additional information in the composition algorithm is an active area of our research.

For more information about the SILO Ontology and the decreased Java Construct Agent, please refer to [39, 40].

3.1.7 SILO Services

Silo services are the fundamental building blocks, as presented in Section 2.2.1. A SILO service exists as a Linux shared-object (*.so) file. Like other dynamic linked libraries, such as a DLL in Windows, it can be loaded to the memory by the SMA when it is needed.

More details about designing SILO services are presented in Section5.1.

3.1.8 SILO Tuning Algorithms and Tuning Strategies Storage

A tuning algorithm is a Linux shared-object (*.so) file. It can be loaded to tune the performance of a silo or a set of silos by the SILO Tuning Agent. The Tuning Strategies Storage (TSS) serves as the repository of tuning algorithms for the STA and can be viewed as part of the USS.

More details about SILO tuning algorithms are discussed in Section 4.2.3, Section 4.3, and Section 5.2.

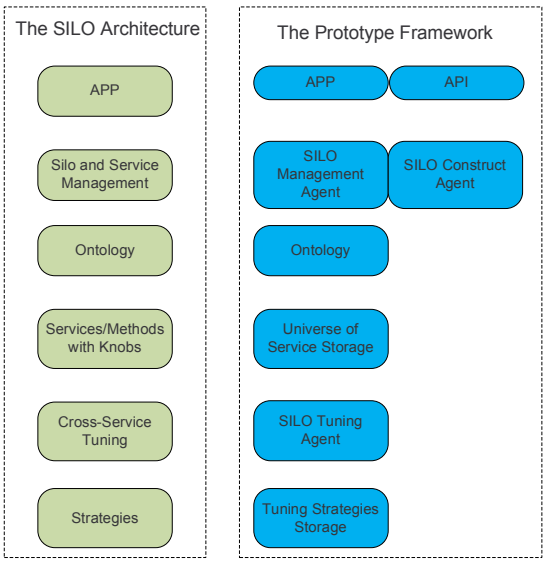

Figure 3.2: SILO Architecture Components vs. Prototype Framework Components

3.1.9 Universe of Services Storage

The USS serves as the main repository of information about the SILO framework. It contains (a) silo services, (b) tuning algorithms, (c) the ontology that describes relationships between services and service interfaces, (d) a database of method implementations that helps the SMA locate the executable code necessary to construct a given silo, and (e) current policy setting that affects the operation of the SILO framework. These can be application-, node- and network-specific.

Silo services and tuning algorithms are presented as *.so files in the USS. Ontology is presented in the format of RDF and RDFS in USS. We chose to implement (d) as an XML file. We call it silo_universe.xml. (e) is our vision and is not implemented in the current prototype.

3.2 Mapping the Prototype Framework to the SILO Architecture

In order to clearly show the relation between the SILO Architecture (shown in Figure 2.2-b) and the SILO prototype framework (shown in Figure 3.1), we put their components side by side in Figure 3.2.

3.3 Silo Setup Procedure and Component Interactions

In this section, we present how the prototype framework responds an application request and compose a silo for it. We also explain component interactions along this procedure.

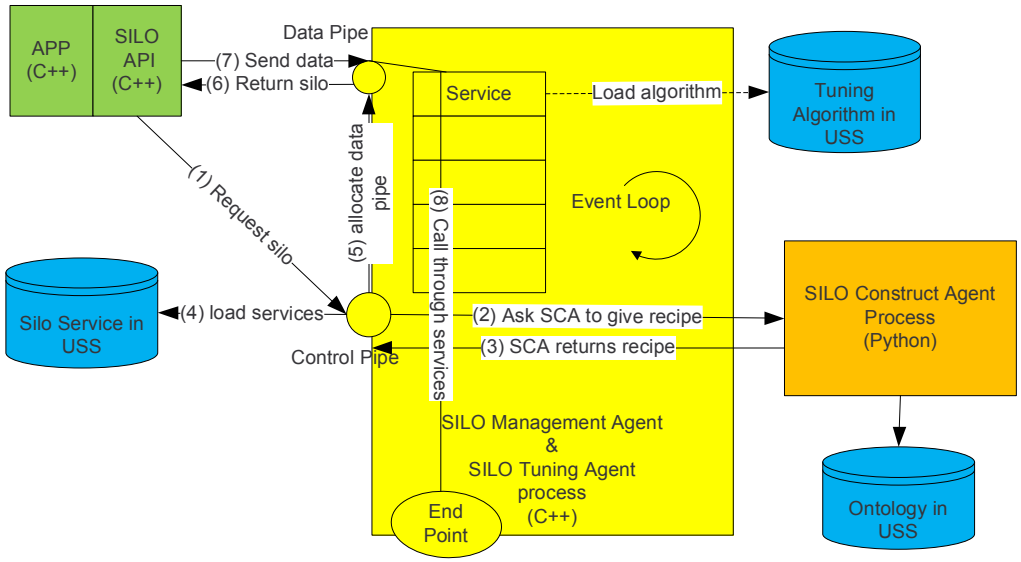

Figure 3.3: Silo Setup Procedure

The procedure is also shown in Figure 3.3. Please follow the sequence number in the diagram.

(1) The application utilizes SILO Application API to create a request, which may include application-customized constraints. This request is forwarded to the SMA by the SILO API through a control pipe in the SMA.

(2) Then the SMA consults with the SCA to validate the request. The communication between the SCA and the SMA is done by Unix sockets.

(3) The SCA validates the request according to the ontology in USS. If the request is valid, the SCA gives a final recipe with a sequence of service names to the SMA.

(4) The SMA uses the returned recipe and silo_universe.xml described in Section 3.1.9 to load service (*.so) files into the memory, keeps the states of every service,

(5) opens the data pipe for further data transmission,

(6) and returns the reference of the newly created silo as a silo handler to the application through SILO application API. The STA could possibly load tuning algorithms from the TSS/USS.

(7) After the application has the proper silo handler, data could be sent to the SMA through the data pipe. Then SMA calls functions in every service to process data. The data exit the SILO framework through the endpoint.

3.3.1 SILO Endpoint

The SILO Endpoint (we may simply refer it as endpoint) is a very important concept we discover when we develop the prototype framework. We envision the SILO architecture as a silo starting from an application and ending at the other end of the silo with the other application. However, for practical reasons, we need to determine where a silo ends in the SILO prototype framework. This is where exactly endpoint is.

Endpoint is where a silo ends, and it is the boundary between the SILO framework and the outside world. As a silo could be bi-directional, technically, an endpoint is also where a silo starts. From this perspective, an application endpoint is needed to connect a silo to an application.

Through endpoints, the SILO prototype framework hands over the control of data to the operating system. Endpoints also collect data and hand the data back to the SILO prototype framework in the opposite direction of the dataflow.

In our current release 0.3, we support two types of endpoints: the UNIX socket endpoint and the UDP endpoint. An application endpoint is a special type of the UNIX socket endpoint.

3.4 Several Implementation Choices

We chose C++ as our main programming language because an object-oriented language fits our architecture design, and C++ can access system resources more freely than Java.

We chose to implement the prototype in the Linux user space because we are focusing on validating the basic architecture concepts and trying to avoid the complexity of kernel space. If it becomes necessary, we may port all the components to the kernel space.

Our prototype is designed to be cross-platform. Most is written in generic C++, although some part is bound with Unix sockets. However, it is only tested under Linux. We anticipate that some effort is needed to port current implementation to another operating system. The decision, officially running only in Linux, helped to significantly accelerate the developing process. We test our prototype against several popular Linux distributions, including Ubuntu, Fedora, Debian and NCSU Realm Linux (a variant of Redhat). We recommend users to run

SILO in Ubuntu and Fedora and give detailed instructions for installing SILO on them in the "SILO User's Guide," which has been released along with the prototype and also appended as Appendix A.

The main component of the prototype, SMA, is an event-driven, not multi-thread program. The greatest advantage we have with the event-driven model is full control of the program in terms of scheduling, which is important when we want to control data flows precisely. The event-driven model also makes it easier to assemble several components while not worrying about synchronization of the components. We do not argue that the event-driven model is necessarily better than the multi-thread model for this prototype, but we chose the former based on the reasons outlined above.

3.5 Software Releases

Since early 2007, we have put much effort into designing the architecture and developing the prototype. As of today (08/2010), there are 253 files, 46,993 lines of code and 156 defined C++ classes with 670 methods in our current developing trunk. There are also six branches dedicated to CSC570, NSF Demo, Virtualization and VCAT extension, TCP-like silo, and IMF.

On 05/02/2008, we released prototype version 0.1. Most of the software components including SMA, SILO API, and Silo Universe were released at that time. A service testing tool was also released, allowing developers to plug in their services into the framework easily.

On 09/25/2008, we released prototype version 0.2. The main changes from 0.1 include multiple bug fixes in the SMA; inclusion of a working tuning agent and sample tuning algorithms; and a working Silo Construct Agent re-written in Python. So far, all defined SILO components have been released.

On 08/01/2010, we released prototype version 0.3. This includes SILO extensions for virtualization and IMF. More SILO services and the SILO Application Gateway were released as well. This release also features an updated users' guide, service programming guide, and other documents detailing some of the source code.

You may download the current release from the SILO official Web site [41].

Chapter 4

Architectural Enabling of Cross-Service Tuning

In this chapter, we explore how we enable Cross-service Tuning in the SILO architecture. We investigate the motivation of Cross-service Tuning in Section 4.1. Then, we explain how we designed the SILO Tuning Agent, knobs and tuning algorithms to enable Cross-service Tuning in Section 4.2. Several tuning algorithms are presented in Section 4.3, and the experimental setup and results are given in Section 4.4.

4.1 Motivation: From Cross-Layer to Cross-Service

We have pointed out the inherent strength of the SILO architecture with respect to cross-layering. As remarked before, cross-layer control is indispensable in many emerging environments, such as wireless multihop networks. However, layering has been a tremendously useful networking paradigm *precisely* because it limits the interaction with the internals of the protocol on one layer with that in another. As a consequence, protocols can be designed and implementations can be written comparatively in isolation, more manageably and leading to more maintainable software. Different protocols for the same layer and different implementations of the same protocol can be "plugged in and out" without affecting the correctness or functionality of protocols at other layers. A cross-layer control algorithm by its nature destroys this useful characteristic because each layer must make some of its internals visible and accessible to other layers. Further, it is rather brittle because changing the protocol at a given layer or even just the implementation of the same protocol may break cross-layer interactions. Thus, the proliferation of cross-layer methods have raised the fear of regression to monolithic software, unmanageable and unmaintainable. Alternatively, cross-layer approaches would remain curiosities and special cases, never to enter significantly the mainstream architecture.

The concept of knobs for services and methods side-steps these problems. The SILO architecture can be viewed as "operate in layers, control across layers." As of today, services and methods are required only to provide a minimal interface, hiding internal details. However, traditional protocols are only required to provide invocation methods (APIs), whereas in the SILO architecture, we require them to provide a minimal control interface as well. Beyond this, the methods can be designed and implemented in isolation, as before. However, the tuning agent has a unique view of the knobs of every method in the silo and can embody all the integrated control concerns. In this way, "cross-service" (the antonym to "cross-layer" in the SILO universe) can become part of the mainstream architecture.

4.2 Realization

In the SILO prototype framework (Figure 3.1), we introduce the SILO Tuning Agent (STA) primarily to perform Cross-service Tuning, as envisioned in the SILO architecture. From the perspective of software, cross-service tuning includes services/methods and their tunable knobs, measurement services/methods and their gauges/read-only knobs, loadable tuning algorithms, and STA. In this section, we present how we implement it by describing how we design knobs and STA.

4.2.1 Design of Knobs

As Figure 2.2 (b) shows, services and methods include tunable interfaces called knobs. However, SILO architecture, or even our prototype framework, does not limit a particular way to design or implement knobs.

This section shows our detailed knob design. We consider the following factors in our designs:

(1) Exposable: Easily expose services' or methods' interfaces;

(2) Tunable: Easily tuned by tuning algorithms; and

(3) Presentable: Easily presented as an understandable form to the tuning agent.

Figure 4.1 presents our knob design. Knobs are typically designed by a service designer. The person who designs a particular service should follow a particular template to design the service's knobs. In the language of C++, a service is a C++ class, and a generic knob is also a C++ class. A particular knob is inherited from the generic knob class and is a member function of a service class. A read-write knob has maximum, minimum, default and current values. The knob may have a tuning step, which is the minimum increment the tuning algorithm is able to

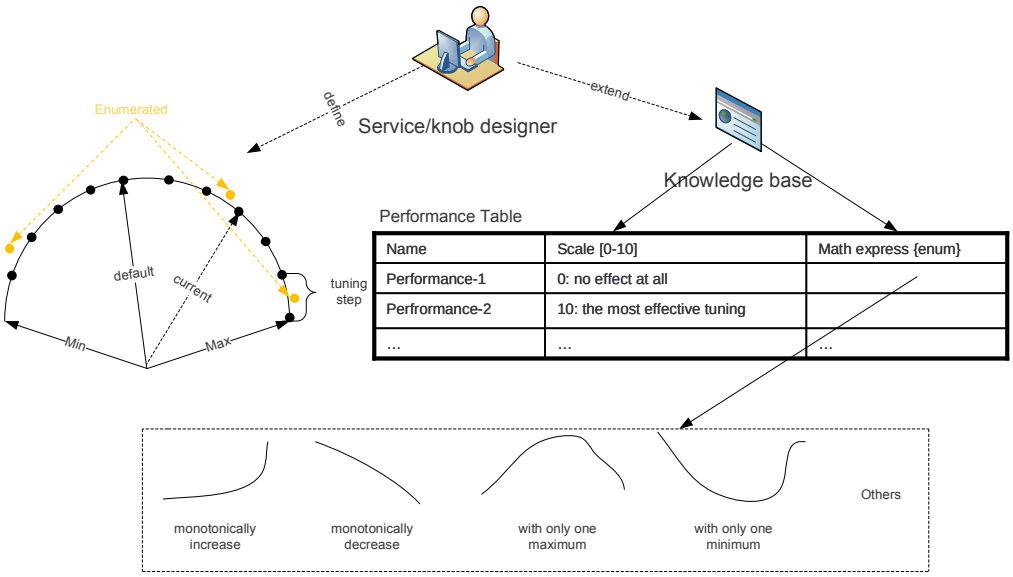

Figure 4.1: SILO Knobs

tune from the maximum to minimum. It is also possible that a knob can be only tuned to a set of predefined enumerated values.

Compared to a read-write knob, a read-only knob/guage is fairly simple and just provides a current value.

Envision Knobs

We also envision certain ways to help expedite the tuning process, although these features are not implemented in the current prototype.

As shown in the right part of Figure 4.1, in addition to the basic properties of a knob, the service/knob developer may provide certain knowledge, such as a performance table. The performance table reveals the correlation between knobs and performances in terms of scales and tuning effects. Some tuning effects can be expressed mathematically.

With the help of this extra knowledge, a proper set of knobs can be more easily selected to reach certain performance optimization. Using a known, mathematically expressed tuning effect may significantly expedite the entire tuning process.

4.2.2 Design of SILO Tuning Agent

The SILO Tuning Agent tunes the knobs of services of a silo or a collection of silos in order to optimize the performance of the silo(s) with respect to a specific objective. This objective may be related to either individual or collective behavior of silos within a single node or among

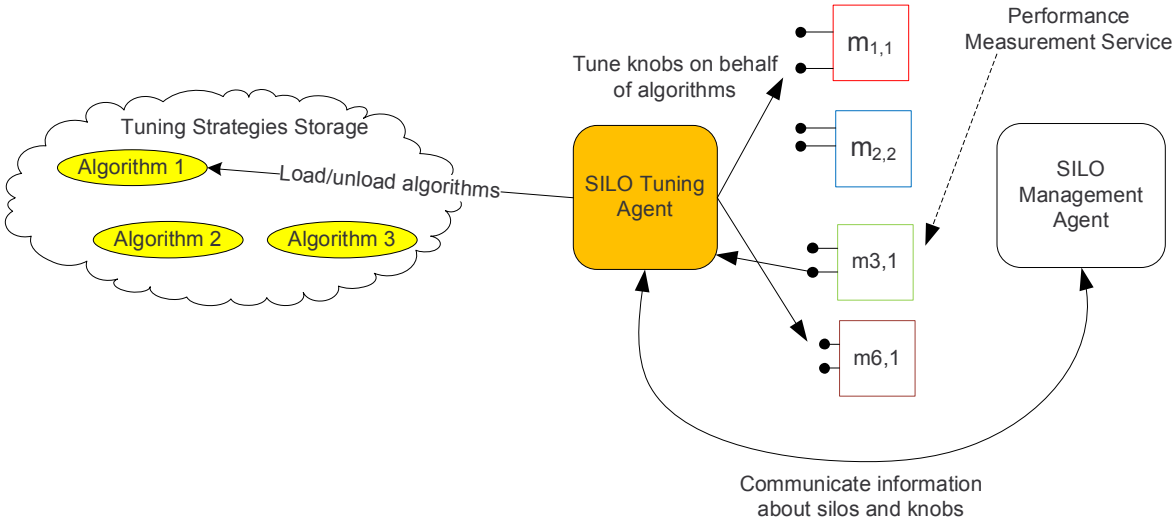

Figure 4.2: SILO Tuning Agent

many nodes. The STA may have at its disposal a range of algorithms or control strategies for adjusting the knobs. The selection of an appropriate control strategy is governed by policies stored within the tuning strategies storage.

Figure 4.2 shows the main responsibilities of the SILO Tuning Agent. The STA loads or unloads tuning algorithms from the Tuning Strategies Storage into the memory; it asks the SMA to pass necessary information in order to enable tuning algorithms, such as a list of silos running in the system and a list of knobs a service/method possesses; it tunes the knobs on behalf of tuning algorithms; it reads back performance measurement from read-only knobs/gauges on performance measurement services, and passes it to tuning algorithms.

4.2.3 Design of Tuning Algorithms

The tuning algorithms are compiled as dynamic link files (*.so files in Linux) and stored in the Tuning Strategies Storage. They can be loaded into the memory by the STA.

A tuning algorithm can be roughly divided into three parts, as follows:

Controller - Asks the STA to tune knobs in a particular way;

Synthesis Rules - Collects performance measurements and decides the overall performance; and

Set Point - Starts or stops the controller when a certain performance level is desired or reached.

We have proposed several tuning algorithms, and they are described in Section 4.3.

4.2.4 Design of Measurement Services

A measurement service is a critical piece in the SILO tuning system. It provides performance information through its read-only knobs or gauges. Some examples of performance measurements are bit error rate, bit per seconds, packet counts, or throughput, etc.

Designing a measurement service is no different from designing other SILO services. The details of designing SILO services are presented in Section5.1.

As the SILO architecture does not specifically define measurement, we investigate the integration of the SILO architecture and the Intergraded Measurement Framework in Chapter8.

4.2.5 Recap the SILO Cross-Service Tuning

As a summary, the SILO Cross-Service Tuning is enabled by the coordination of the following components: knobs, the SILO Tuning Agent, tuning algorithms, and measurement services.

From the perspective of control theory, the SILO Cross-Service Tuning is a self-tuning, closed-loop control system. It equips four basic concepts of a typical self-tuning system: expectation, measurement, analysis, and action.

4.2.6 Envision Cross-Service Tuning

Besides what we have enabled, we also envision certain advanced features and extensions of Cross-Service Tuning, although they are not implemented in the current prototype framework. Figure 4.3 shows our grand vision.

Cross-Node Tuning Cross-node tuning expands the cross-service tuning from within a single node to multiple nodes. With a better understanding of the network, STAs are able to provide more precise tuning. After the silo connection is set up, the STA in the local node can communicate and coordinate with the STA in the remote node along with STAs in the path between the local node and the remote node. SYNC in Figure 4.3 refers to the crossing-node tuning synchronization and to the coordination between the local node and the remote node. Cross-node tuning also could take into consideration the disturbance from other silos in the Internet along the data flow path.

Knowledge Base Aided Tuning Experiences of good tuning practices could be stored in the knowledge bases. The knowledge bases could reside either locally or globally. Good practice can always be propagated to the global knowledge bases. By using the help of the knowledge base, the time cost and results of tuning could be significantly improved.

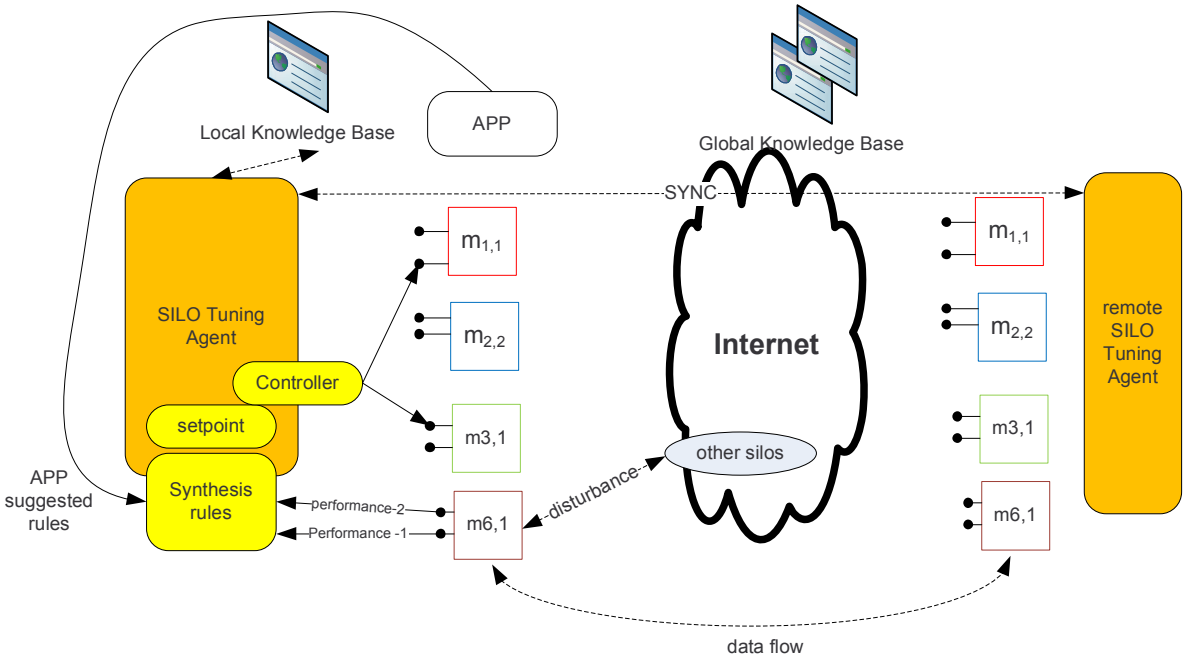

Figure 4.3: Cross-Service Tuning Vision

Application Suggested Rules SILO-enabled applications should be able to suggest the set point and the synthesis rules. Applications' suggestions are subject to approval by the SILO Tuning Agent. It gives applications greater flexibility, and enables the SILO architecture to better serve applications.

4.3 Tuning Algorithms

In this section, we present some tuning algorithms and explore ways for them to evolve into advanced tuning algorithms.

4.3.1 Problem Notations and Abstraction

We introduce our notations below, which are used in all the algorithms. K is the number of knobs. For knob i, its value is $V_i(i : 1 \rightarrow K)$; it has m_i possible values, denoted as $V_{i,1} \rightarrow V_{i,m_i}$ with the tuning step S_i (or enumerated values). The number of searches is N. Time for testing a combination is T. Our objective is to find an unknown function's maximum, minimum, a value larger or smaller than a certain threshold, or a combination of certain criteria.

We define the **distance** of two knob value combinations as the number of steps that can tune one combination to another combination. If the distance is 1, these two combinations are

neighbors to each other.

4.3.2 Basic Algorithms

The pseudo code is given for most of the basic algorithms. We use the maximum performance as the criterion.

Exhaustive Search

Exhaustive Search examines all possible combinations of knob values, then tunes the knobs to the combination that provides the optimal performance.

Visit Complexity: $\prod_{i=1}^{K} m_i$

Basic Random Search

Basic Random Search randomly examines some combinations of knobs and obtains an approximate maximum after several tries.

$Basic\_Random\_Search(K, N)$
$max \leftarrow 0$
for $n \leftarrow 1$ to N
  for $i \leftarrow 1$ to K
    randomly select from $V_{i,1}$ to V_{i,m_i}
  $result \leftarrow performance$(selected knobs)
  if $result > max$
  then $max \leftarrow result$
    for $i \leftarrow 1$ to K
      $V_i \leftarrow$ selected knobs i
return max

Visit Complexity: N

$Basic\_Random\_Search()$ can be used to find the starting position for advanced tuning algorithms, which is explained later in this section.

Greedy Search

Greedy Search tunes one knob at a time. After it sets a knob, this will not be adjusted again.

$Greedy\_Search(K)$
for $i \leftarrow 1$ to K
  $max \leftarrow 0$
  $V_i \leftarrow 0$
  for $j \leftarrow 1$ to m_i
    $result \leftarrow performance(V_{i,j})$
    if $result > max$
    then $max \leftarrow result$
      $V_i \leftarrow V_{i,j}$
return $performance$(knobs' position)

Visit Complexity: $\sum_{i=1}^{K} m_i$

4.3.3 Strategies to Evolve to Advanced Algorithms

Reduce the Number of Visits

(1) Combine basic algorithms to achieve better approximation;

(2) Use proven-good or average-good approximation algorithms to search for the approximation in a limited effort, such as Simulated Annealing, etc.;

(3) Skip or rule out some knobs that are less relevant for certain criterion.

Use Meaningful Criteria

If the performance meets the already-known maximum or minimum, the algorithm stops immediately. For instance, if the criterion is packet loss rate, packet error rate or retransmission rate, the algorithm stops immediately once the performance testing result is 0 or 1.

4.3.4 Improved Greedy Search

This algorithm tries Greedy Search from several different starting combinations and different sequences of knobs, and returns the best performance it ever encounters. If there is only one starting point, it is equal to greedy search.

$Improved\_Greedy\_Search(K, N)$
$max \leftarrow 0$
for $n \leftarrow 1$ to N
  for $i \leftarrow 1$ to K

randomly select from $V_{i,1}$ to V_{i,m_i}
randomly rearrange the sequence from Knob i to Knob K
if $Greedy\_Search(K) > max$
then $max \leftarrow Greedy\_Search(K)$
return max

Visit Complexity: $N * \sum_{i=1}^{K} m_i$

4.3.5 Hill Climbing Search

Hill Climbing always selects the neighboring combination with the best performance. When the performance of all neighboring combinations worsens, this local maximum is returned.

$Hill\_Climbing\_Search()$
$currentComb \leftarrow startComb$
loop do
  $L \leftarrow neighbors(currentComb)$
  $nextEval \leftarrow -\infty$
  $nextComb \leftarrow \emptyset$
  for all x in L
    if $(performance(x) > nextEval)$
      $nextComb \leftarrow x$
      $nextEval \leftarrow performance(x)$
  if $nextEval <= performance(currentComb)$
    return $currentComb$ //Return current combination because no better neighbors exist
  $currentComb \leftarrow nextComb$

4.3.6 Clustered Random Search

The process of algorithm is explained as follows.

$Clustered\_Random\_Search(K, C_1, C_2, N_1, N_2)$

(1) First obtains C_1 maximums as max-candidates by $Basic\_Random\_Search(K, N_1)$ or $Improved\_Greedy\_Search(K, N_1)$;

(2) Randomly selects values (N_2 times) for knobs around these candidates, and keep the distance less than C_2; and

(3) Returns the max.

Visit Complexity: $N_1 + C_1 * N_2$

4.3.7 Simulated Annealing Search

Simulated annealing is a general probabilistic meta-heuristic for optimization problems, which can be easily adapted for SILO tuning.

$Simulated\_Annealing\_Search(P, N, criterion)$

$s \leftarrow s_0; e \leftarrow performance(s)$	// Initial state, energy.
$i \leftarrow 0$	// Energy evaluation count.
while $i < N$ and $e < criterion$	// While time remains & not good enough
$s_i \leftarrow neighbor(s)$	// Pick some neighbor.
$e_i \leftarrow Performance(s_i)$	// Compute its energy.
if $P(e, e_i, temp(i/N)) > random()$ then	// Should we move to it?
$s \leftarrow s_i; e \leftarrow e_i$	// Yes, change state.
$i \leftarrow i + 1$	// One more evaluation done
return s	// Return current solution

Visit Complexity: N
When we try to find the maximum performance, the criterion is ∞.

4.4 Experimental Setup and Results

4.4.1 Testing Scenario

Figure 4.4 explains the testing scenario, which includes two tuning agents and two silos. We purposely ignore other components of the SILO architecture in order to give a clearer picture in terms of tuning. Each silo in Figure 4.4 can reside in different machines or the same machine.

There are four services/methods in each silo. In this example, methods are the only instances of corresponding services and bear the same name, so we do not distinguish between services and methods in this example and call them services for convenience.

The RTX (Retransmission) service sends data to a lower service and waits at most for RTX:$time\_out$ to determine whether it needs to retransmit the data; it also maintains a sliding window with the size of RTX:$window\_size$. The ACK service generates packet sequences responds ACK for every ACK:$frequency$ packets (e.g., if frequency is 1, ACK responds to every packet, and if frequency is 2, ACK responds to every other packet), or wait for at most ACK:$wait\_time$. The performance service is able to counter packets with a read-only $Performance$:$pkt\_counter$. The UDP service adds or removes the UDP header, and it is tunable by UDP:$port$.

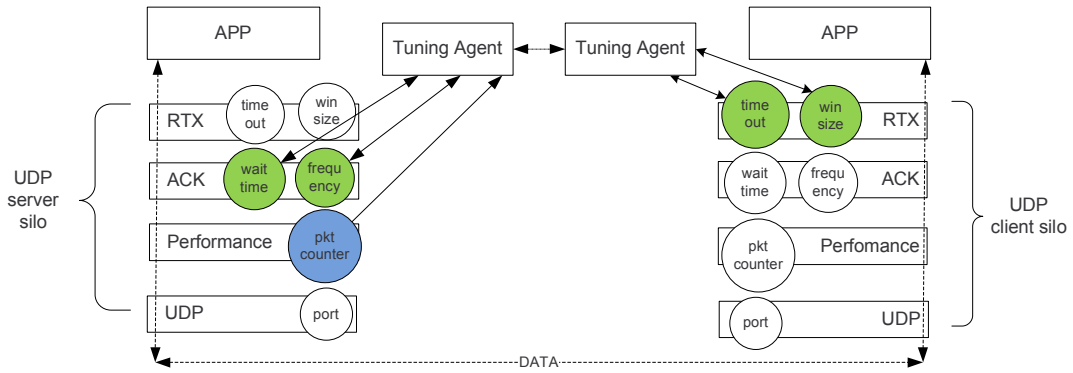

Figure 4.4: Tuning Agent Testing Scenario

The working process is explained below. Firstly, the tuning agent asks a silo for its service list by the silo handler (or silo ID). Secondly, the tuning agent asks each service to return its knob list. Finally, with all the information of knobs from a silo, the tuning agent is able to communicate with another tuning agent, to collect all knob information from two silos, to select proper algorithms and strategies, and to obtain the desired performance by tuning.

4.4.2 Result

We provide results in two different scenarios. In the first simple scenario, only the tuning agent on the UDP server side tunes two knobs on ACK service. ACK:$wait\_time$ goes from 0ms to 40ms with the tuning step of 10ms; ACK:$frequency$ is tunable from 1 to 5 and the tuning step is 1. RTX:$window\_size$ is set to 1, which means there is only one buffer available, and the system behaves like a stop-and-wait ARQ system. RTX:$time\_out$ is set to 50ms to eliminate necessary retransmission.

In the second complicated scenario, two tuning agents coordinate to tune four knobs. On the UDP server side, ACK:$wait\_time$ goes from 0ms to 40ms with the tuning step of 10ms; ACK:$frequency$ is tunable from 1 to 5 and the tuning step is 1. On the UDP client side, RTX:$time\_out$ goes from 10ms to 50ms with a tuning step of 10ms; RTX:$window\_size$ can be tuned from 1 to 5, and the tuning step is 1.

Figure 4.5 shows that all 25 possible knob combinations are visited for the simple scenario. Figure 4.6 shows that all 625 possible knob combinations are visited in the complicated scenario. We provide a third result for exhaustive search, which doubles the possible values for each knob from 5 to 10. Therefore, Figure 4.7 shows that all 10,000 possible knob combinations are visited. The accurate maximum performance is always identified. However, because all possible combinations are visited, it requires the greatest amount of time.

Figure 4.8 shows that 12 out of 25 random knob combinations are visited in the simple

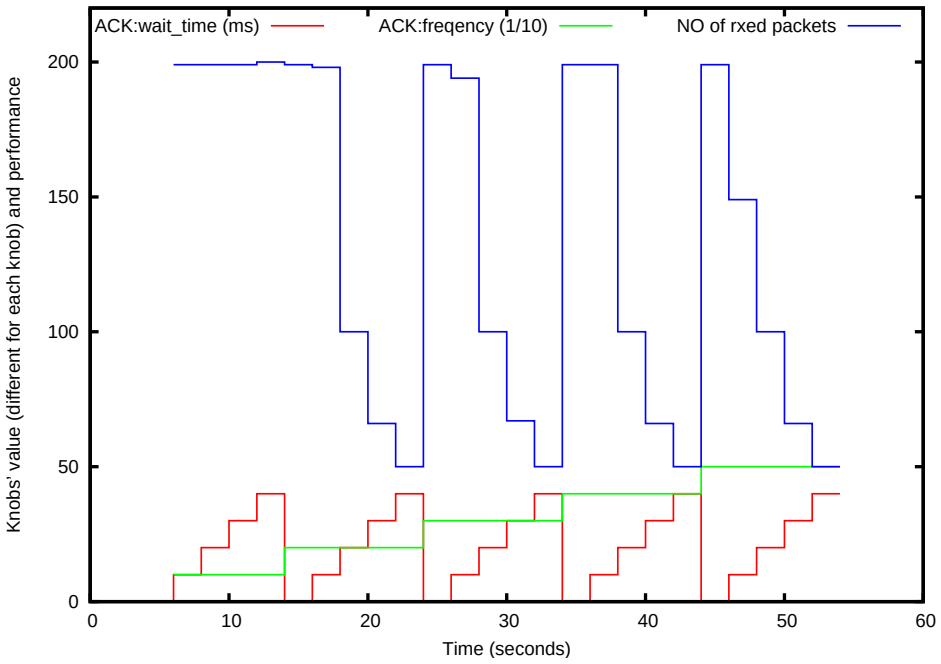

Figure 4.5: Exhaustive Search with 2 Knobs Tuned

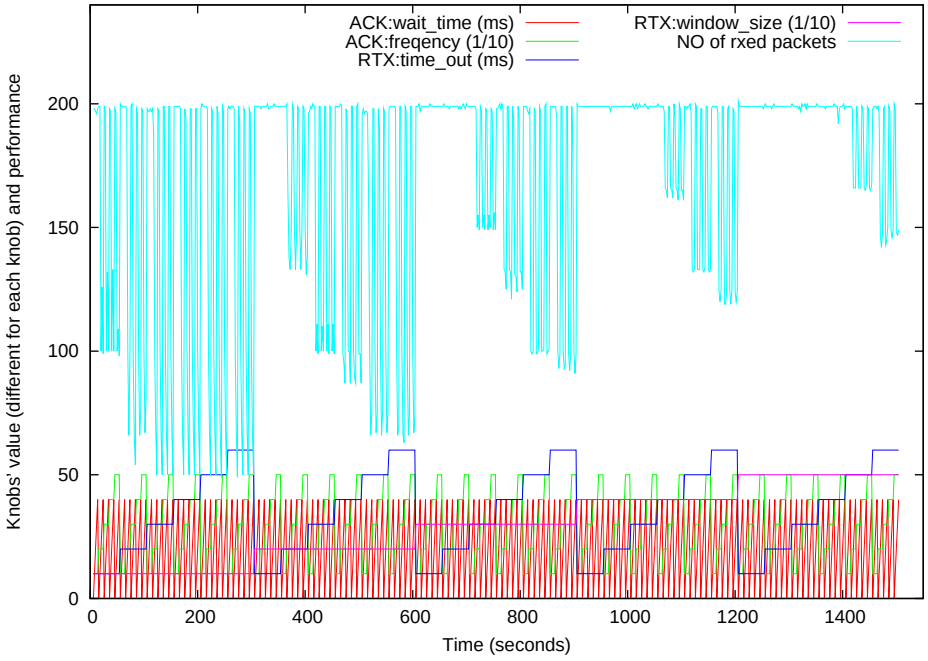

Figure 4.6: Exhaustive Search with 4 Knobs Tuned

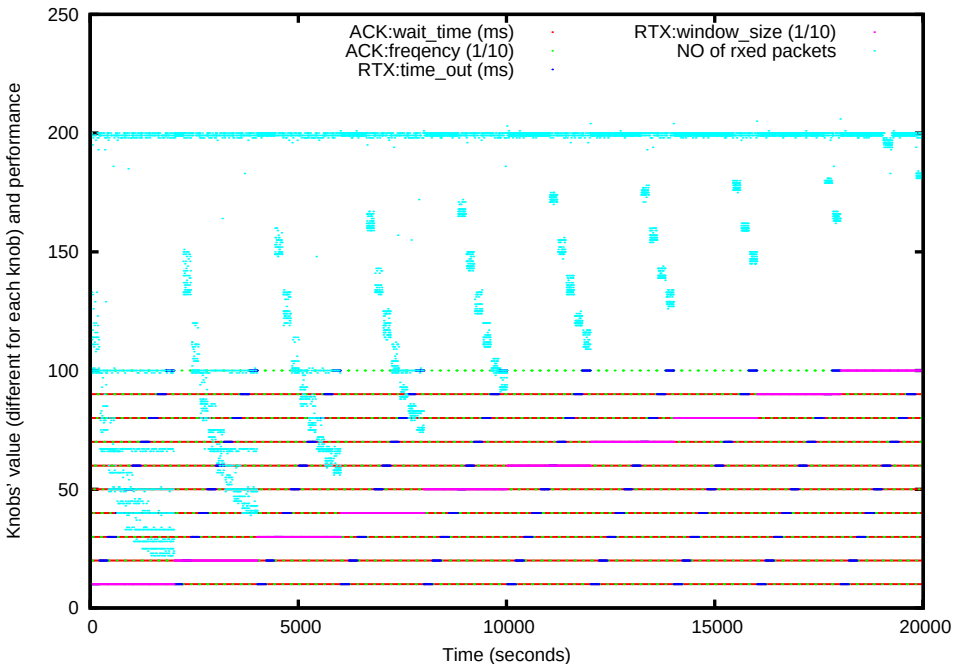

Figure 4.7: Exhaustive Search with 4 Knobs Tuned and Doubled Up

scenario. Figure 4.9 shows that 80 out of 625 random knob combinations are visited in the complicated scenario. From the result, the maximum performance is found, but it is not always guaranteed.

Figure 4.10 illustrates the improved greedy search in the simple scenario, which includes two greedy searches. Figure 4.11 shows the improved greedy search in the complicated scenario, which also includes two greedy searches. From the result, the maximum performance is likely found, but due to the restriction of greedy search, it might fall into the local significance.

Figure 4.12 illustrates the hill climbing search in the simple scenario. Figure 4.13 shows the hill climbing search in the complicated scenario. From the figures, the hill climbing search quickly reaches the maximum performance and stays in that level.

Figure 4.14 illustrates the clustered random search in the simple scenario. Figure 4.15 shows the clustered random search in the complicated scenario. Like random search, although it identifies the maximum performance, it is not always guaranteed.

Figure 4.16 illustrates the simulated annealing tuning process in the simple scenario. Figure 4.17 illustrates the simulated annealing tuning process in the complicated scenario. From a random starting point, the algorithm finally converges to maximum performance.

In conclusion, the algorithms demonstrate the trade-off between time and accuracy. From our results, hill climbing and simulated annealing provide a good balance of trade off and give

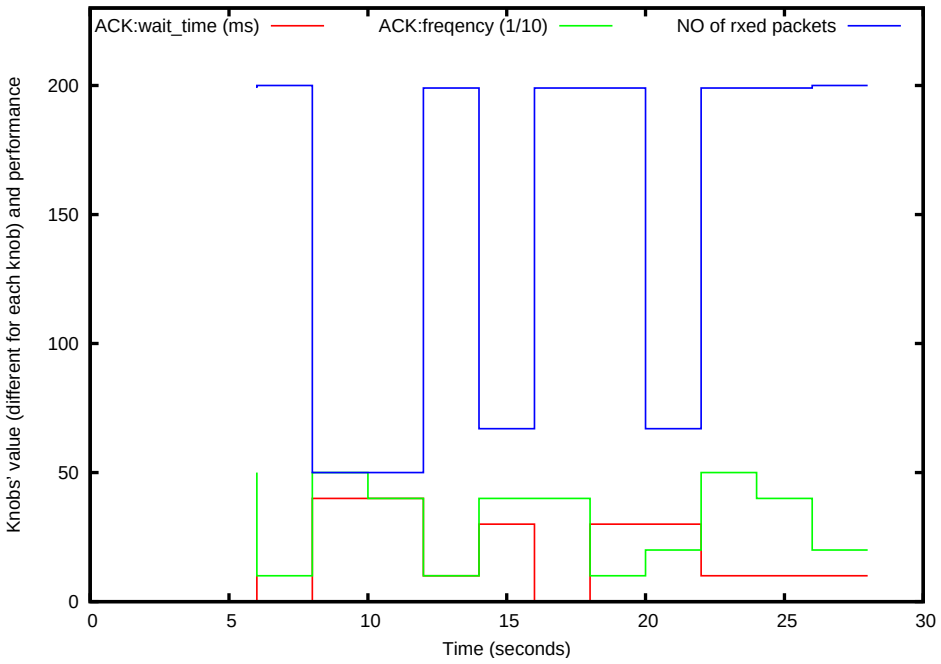

Figure 4.8: Basic Random Search with 2 Knobs Tuned

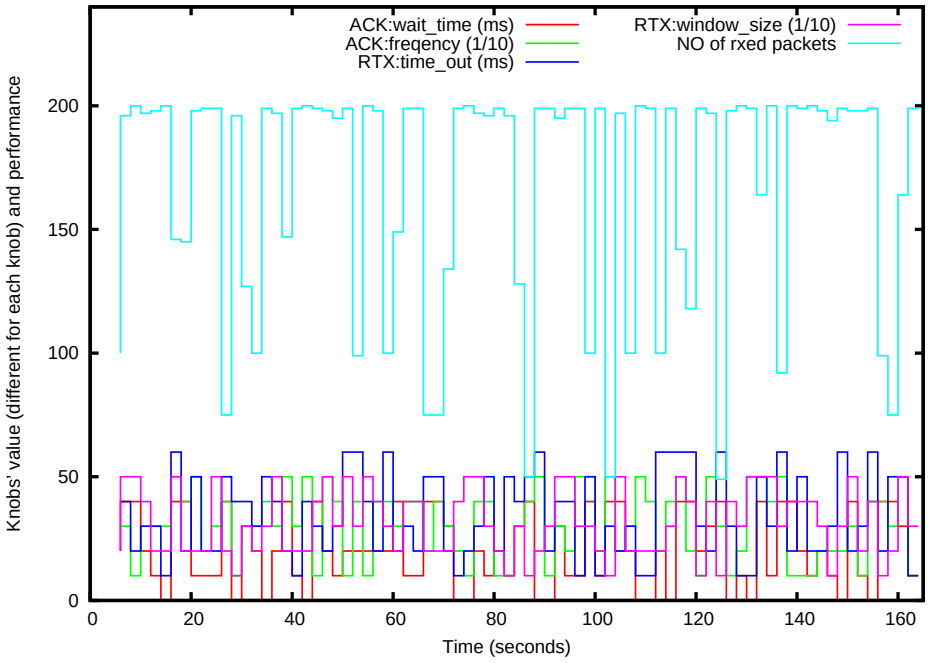

Figure 4.9: Basic Random Search with 4 Knobs Tuned

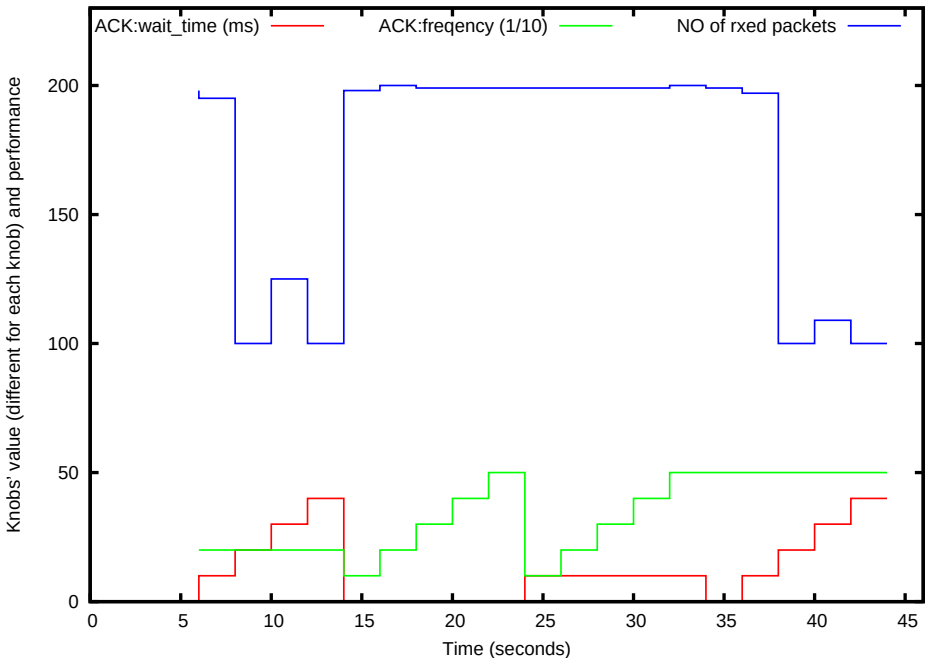

Figure 4.10: Improved Greedy Search with 2 Knobs Tuned

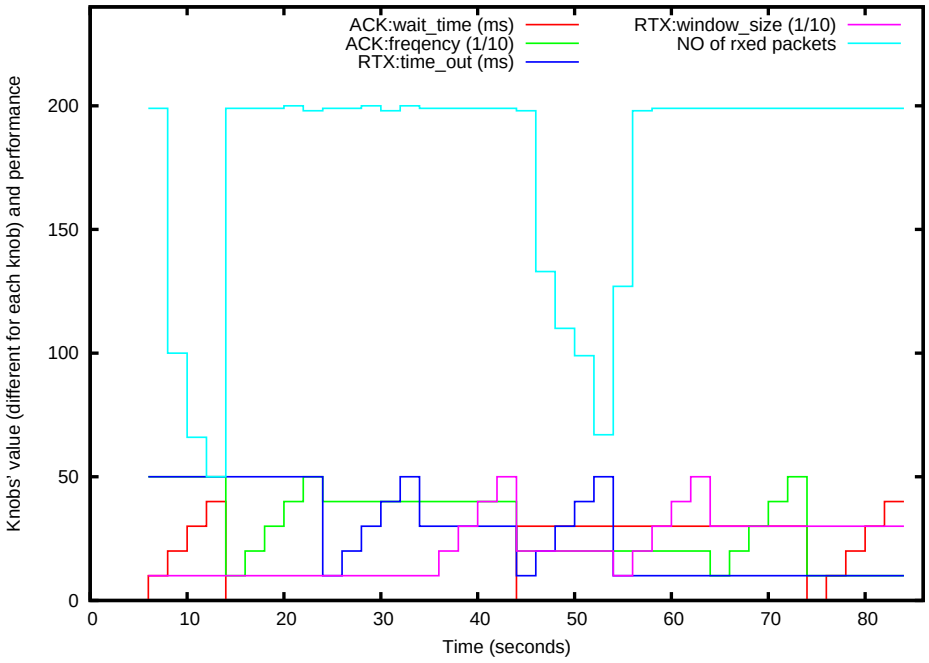

Figure 4.11: Improved Greedy Search with 4 Knobs Tuned

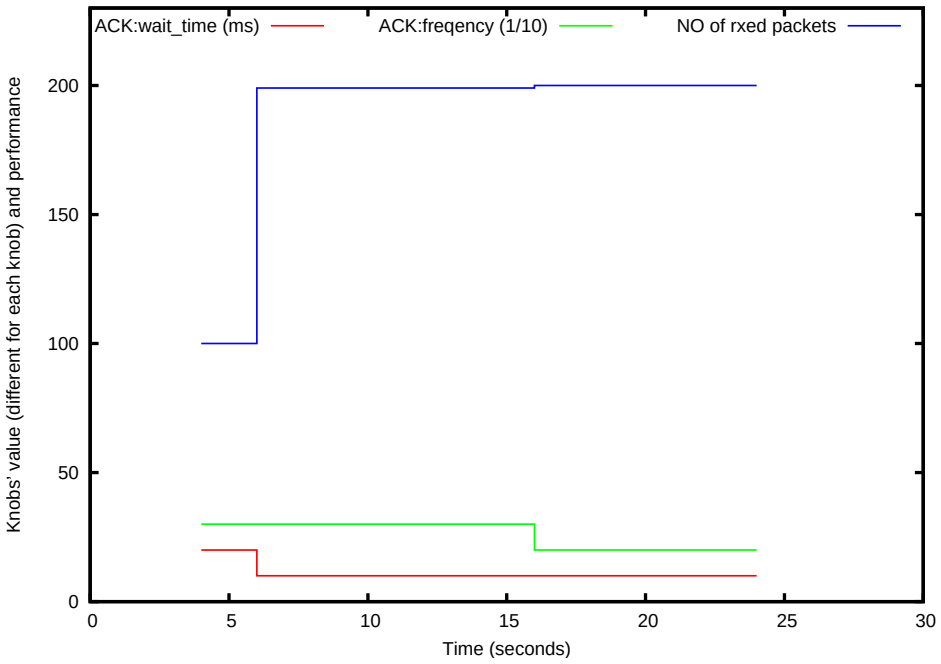

Figure 4.12: Hill Climbing Search with 2 Knobs Tuned

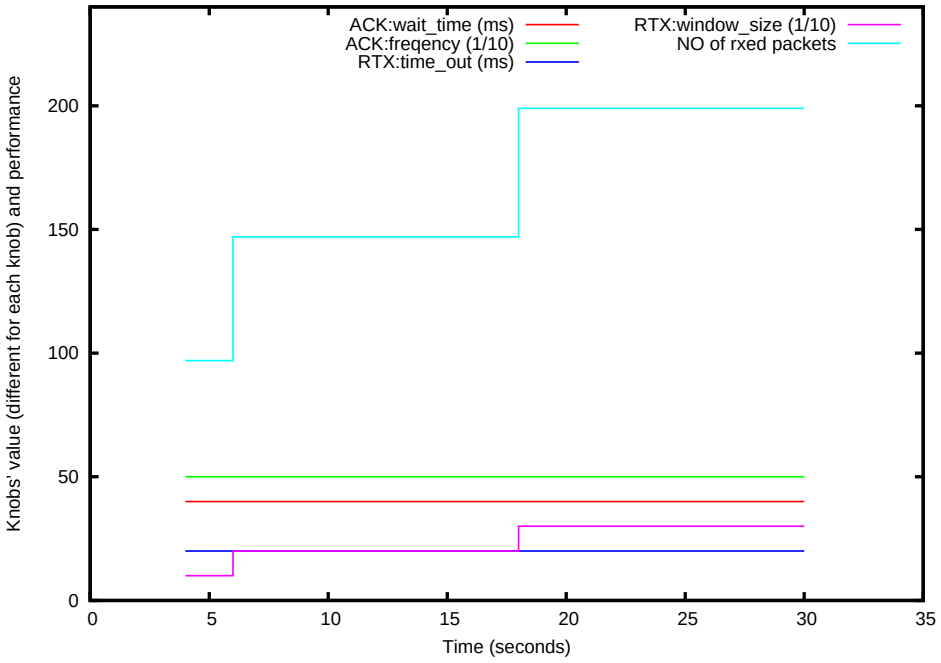

Figure 4.13: Hill Climbing Search with 4 Knobs Tuned

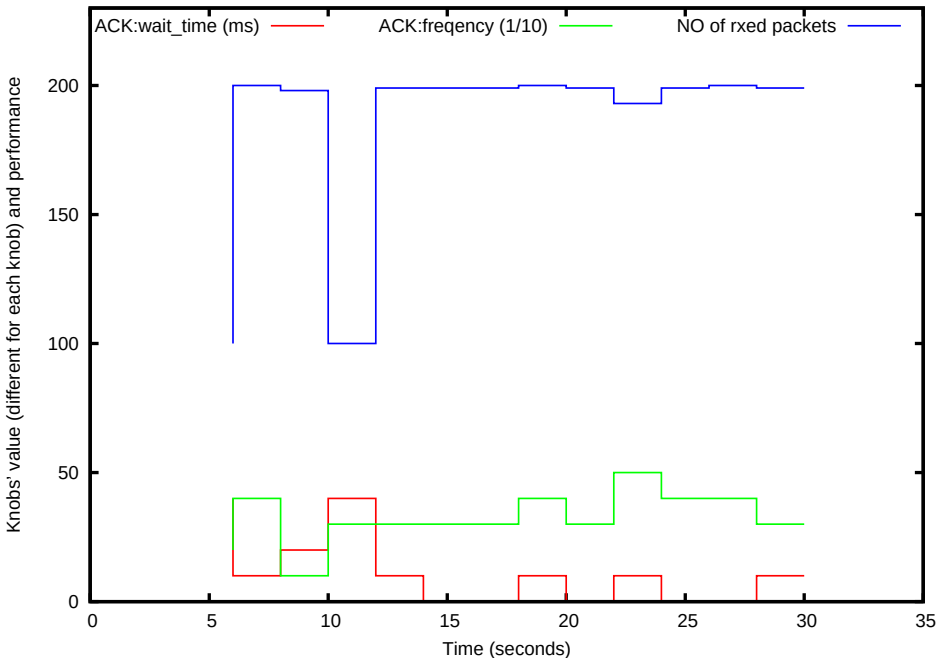

Figure 4.14: Clustered Random Search with 2 Knobs Tuned

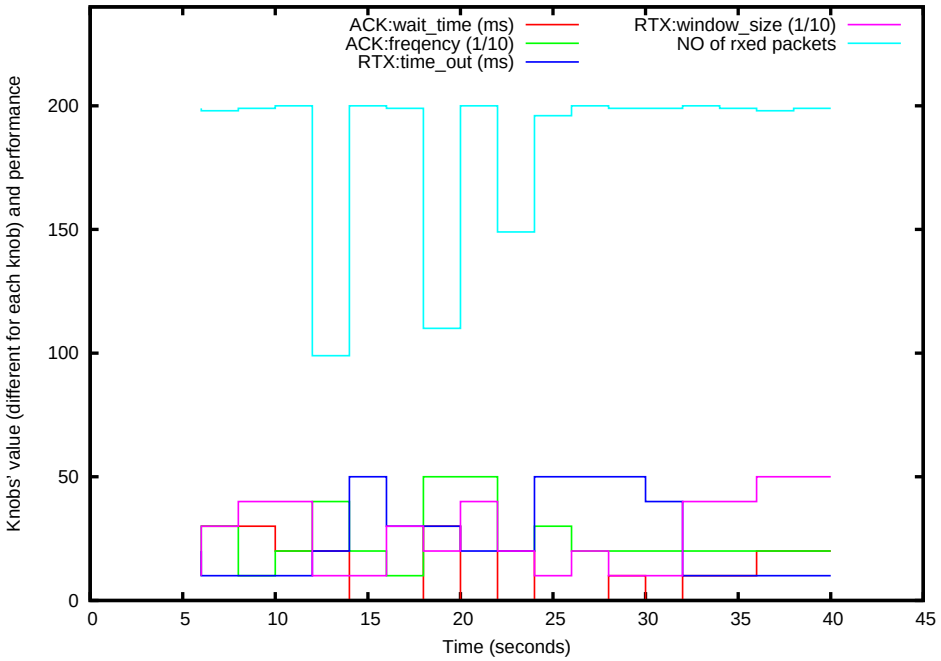

Figure 4.15: Clustered Random Search with 4 Knobs Tuned

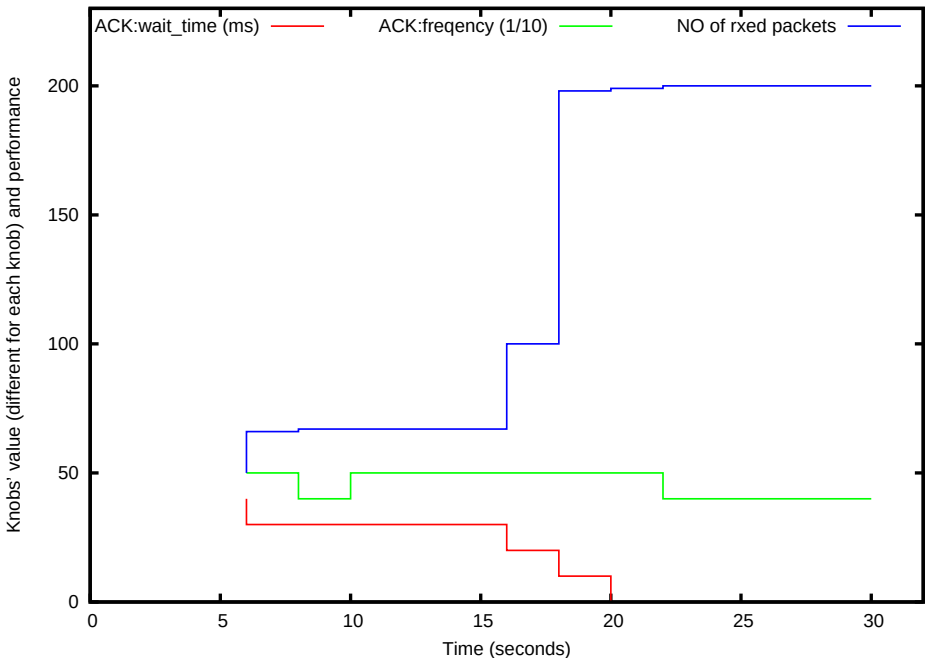

Figure 4.16: Simulated Annealing Search with 2 Knobs Tuned

relatively accurate findings within less time and less cost.

In the next chapter, we examine another important aspect enabled by the SILO architecture: programming in the SILO architecture.

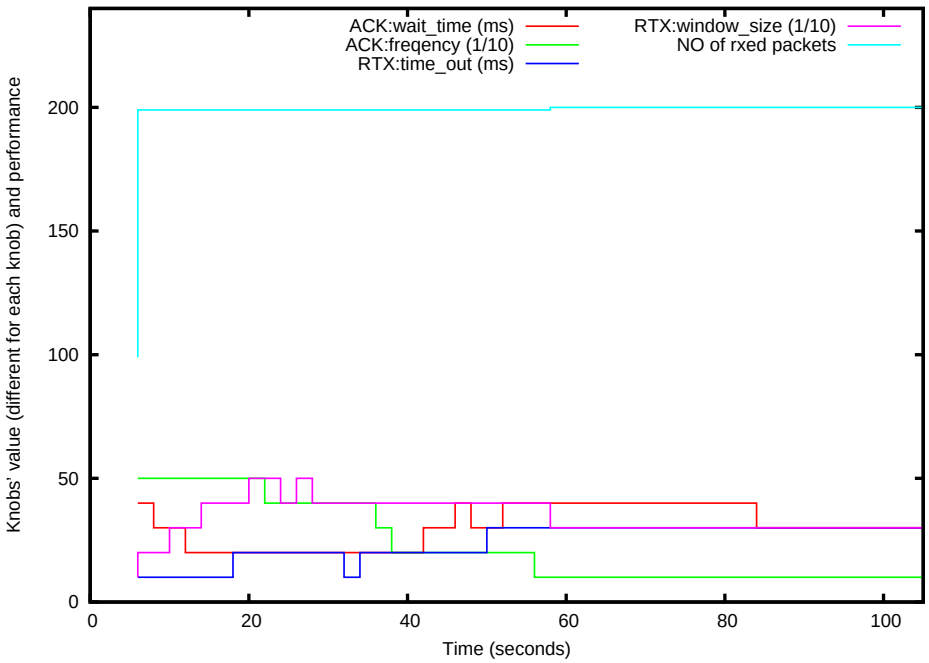

Figure 4.17: Simulated Annealing Search with 4 Knobs Tuned

Chapter 5

Architectural Support for Customized Design in SILO

In Section2.1, we articulate our design philosophy for the SILO architecture - *designing for change: it is precisely the characteristics of the system that do not change themselves, but provide a framework within which the system design can change and evolve.*

In this chapter, we present our architecture support for customized design in SILO. This is a fundamental reflection of our design philosophy. Our architecture enables users to contribute by designing their own services and tuning algorithms. It also gives users greater flexibility to design SILO applications, through which application-customized silos are able to obtained.

5.1 Designing SILO Services

In Section 2.2.1, we present the full definition of services by describing: (1) the function it performs, (2) the interfaces it presents to other services, (3) any properties of the service that affect its relation with other services (e.g., as required to establish a partial ordering), and (4) its knobs and their actions and constraints.

In the SILO prototype framework, we provide the sample service template to guide service programmers to implement the above four requirements, and also provide make files to compile services to *.so files. For the details of service programming, please refer to the "SILO Development Guide" in the SILO prototype release 0.3.

5.1.1 Service Programming Case Study

Is there any evidence to show that SILO service programming lowers the barrier to continuing innovation, its stated goal? Of course, the answer to such a question would require a long

and diverse experimental effort, and to be convincing, would have to come at least partly from actual developer communities after at least partial deployment.

However, we have been able to conduct a small case study that has yielded encouraging results. In the fall of 2008, we made a simplified version of the SILO codebase (shown in Figure 5.1) available to graduate students taking the introductory computer networks course at North Carolina State University. Students are encouraged to take this course as a prerequisite to advanced graduate courses on networking topics, and most students taking the course have no prior networking courses, or even a single undergraduate course on general networking topics. Students are required to undertake a small individual project as one of the deliverables, which typically involves conducting a literature research on a focused topic and synthesizing the results in a report. In this instance, students were offered the option to attempt to program a small networking protocol as a SILO service as an alternative project. Nine out of the approximately fifty students in the class chose to do so. All but one of these students had not coded any networking software previously.

To our satisfaction, all nine produced code to perform non-trivial services, and the code not only worked, but it was possible to compose the services into a stack and interoperate them, although there was no communication or effort among the students to preserve interoperability during the semester. In one case, the code required reworking by the teaching assistant because the student concerned had (against instructions) modified the SILO codebase distribution. The services coded by the students were implementations of ARQ, error control, adaptive compression, rate control, and bit stuffing. Testing services such as bit error simulators were also coded, and two students attempted to investigate source routing and label switching, going into the territory of SILO services over multiple hops, which are as yet comparatively unformed and malleable in our architectural vision.

While this is only the very beginning of attempts to validate SILO, we feel that this case study at least demonstrates that the barrier to entry into programming networking services has been lowered, in that the path from conceptual understanding of a networking protocol function to attaining the ability to produce useful code for the same is dramatically shorter. In future similar case studies, we hope to study the reactions of such beginning programmers to the tuning agent and ontology capabilities. Furthermore, as always, we continue to invite the community to download the SILO code from our project Web site [41], to experiment with it, and to notify us of their positive and negative experiences.

In order to give students full concentration of services, we provide a simplified SMA to the course project, which is shown in Figure 5.1. Compared to Figure 3.3, Figure 5.1 is a much simplified SMA. We disable the tuning agent and tuning algorithms; then, we replace the SILO Construct Agent with a simple xml file – recipe.xml. We also integrate an application inside the SMA, so the whole project is able to run in one process. This benefits the students and

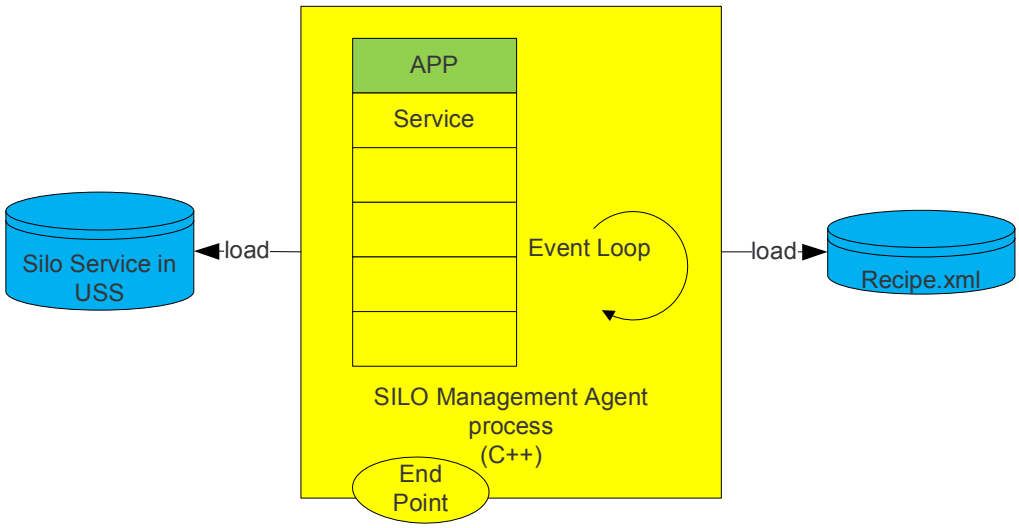

Figure 5.1: Simplified SMA for CSC 570 Networking Course

Table 5.1: 2008 Fall CSC 570 SILO Projects (by permission of the students)

Names	**Project Title**
Ahmet Can Babaoglu	Error Control and Simulator for SILO Project
Aman Nijhawan	Programming Go Back N Service for Silo
Abhivav Naik	Adaptive Huffman Compression Service for Silo Framework
Hiran Mayukh	Adaptive Compression Service by Token Bucket for Silo Framework
Kumar Gokare	Implementation of Selective Repeat ARQ Service in Silo
Ritesh Anand	Bit Stuffing Service under Silo
Vikram Mulukutla	Source Routing
Zainab Vora	Implementation of Label Switching in Silo Environment

saves their time, so they may better understand a complicated architecture.

We will repeat the same exercise in the near future. With the more mature SILO prototype framework, we are ready to provide the standard codebase to students.

Table 5.1 gives a brief overview of each student's project.

For more details regarding the SILO lab, please refer to Appendix B.

5.2 Designing SILO Tuning Algorithms

In Section 4.2.3, we present the main components of a tuning algorithm. Then, in Section 4.3, we discuss some sample tuning algorithms.

Similar to programming SILO services, we provide the sample tuning algorithm template to guide tuning algorithm programmers to implement them, and we also provide make files to

compile tuning algorithms to *.so files. For the details of tuning algorithm programming, please refer to the "SILO Development Guide" in the SILO prototype release 0.3.

5.3 Designing SILO Applications

In the SILO architecture, SILO applications are provided greater flexibility to request customized silos with application-defined constraints or ontology. We also envision that applications should be able to suggest tuning strategies.

In the SILO prototype framework, we provide the SILO application API to let programmers create customized constraints or requests, to request and release a silo, to send and receive data from the created silo.

For more details about SILO application programming, please refer to the sample application and the "SILO API" document in the SILO prototype release 0.3.

In the rest of this section, we present two special SILO applications we have built: the SILO Application Gateway (SAG) and a complete SILO application with a graphic user interface for NSF demonstration purposes.

5.3.1 SILO Application Gateway

SILO applications call the SILO Application API to make use of the SILO architecture. This indicates that many of the already existing applications have to be rewritten and recompiled with compliance to the SILO Application API.

In order to provide an easy way to use some popular applications with SILO architecture, we provide the SILO Application Gateway (SAG). It is a special application designed to bridge some popular applications and SILO architecture.

Figure 5.2 shows how the SILO Application Gateway fits the SILO architecture. Some popular media player applications, such as the VLC Media Player, can send a UDP stream to the SILO Application Gateway. The SILO Application Gateway then calls the SILO API and forward the UDP packet stream to a silo. After the stream goes through the silo, it is transferred as a SILO packet stream to the other side. Finally, the stream reaches the VLC Media Player on the other side, so that the stream can be rendered.

5.3.2 NSF Demos

On 04/07/2009, we successfully demonstrated the SILO prototype framework at an NSF FIND project meeting. This is the first SILO GUI application that supports all basic SILO-enabled features.

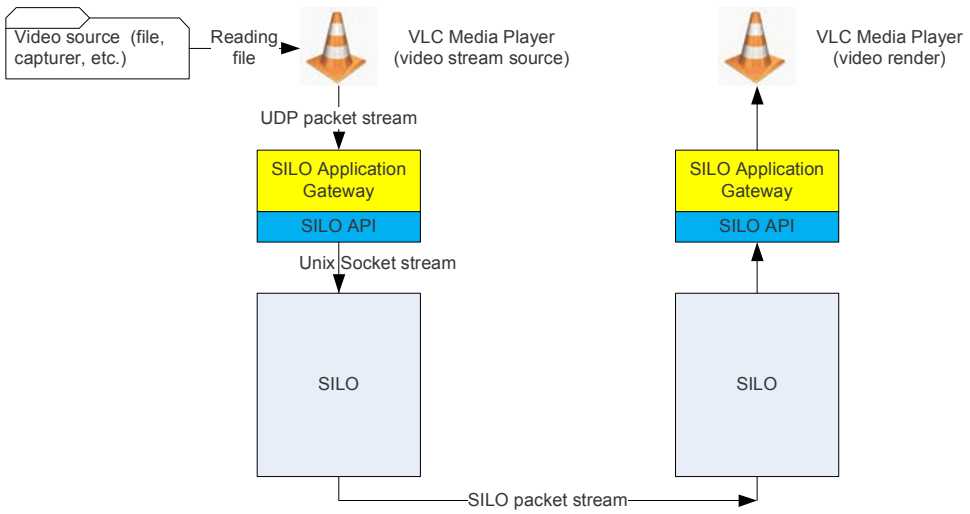

Figure 5.2: SILO Application Gateway

A SILO-enabled application is featured on the left side of Figure 5.3. The five-step procedure outlines how a typical SILO-enabled application works. In the first and second steps, the application can build its own requirements and constraints. These are the extra constraints, which will in total become the ontology constraints. In the first step, required services can be chosen. In the second step, applications are able to request a particular sequence of services. Step Three will request the SMA to create a silo, which can be used by the application. After the application has access to the silo, it may use the silo to send and receive data before requesting the SMA to release the silo during the last step.

The right side of Figure 5.3 is a GUI for the tuning agent. Although the tuning agent is not supposed to have a GUI, this is built for the demo to give a clear view of how the tuning agent interacts with SILO services.

If the Token Bucket service is chosen to be included in a silo by the application, we may click the radio button on the tuning agent's GUI side to manually tune the token rate of the token bucket service. The token rate is a knob defined in the token bucket service. We finally collect data and use the gnuplot tool to plot the data.

The log box on the right side displays necessary information to users. It provides information about the final recipe decided by the SCA, and about the silo id that has been created by the SMA.

As seen in Figure 5.4, the rate of sending packets changes according to the token rate of bucket service.

So far, we have finished introducing our basic design of the SILO architecture. In the next

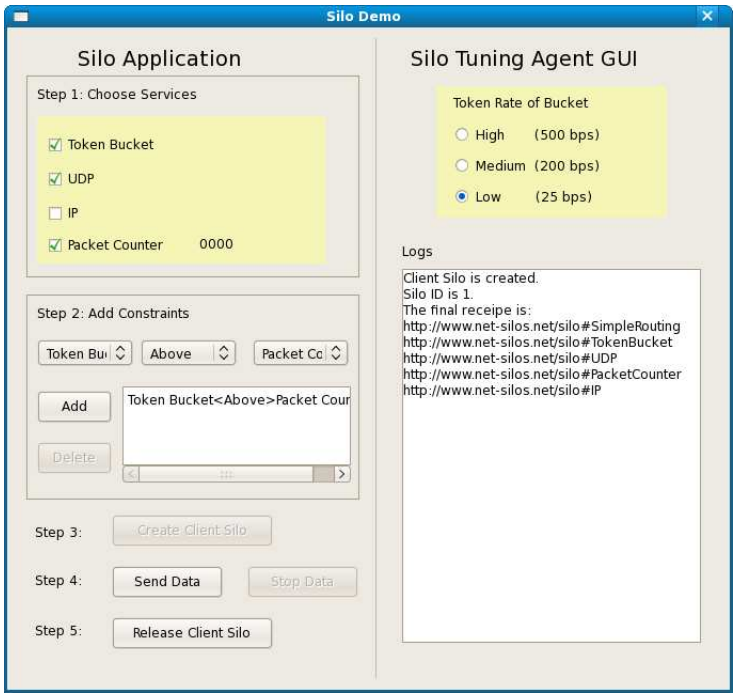

Figure 5.3: NSF SILO Demo GUI

three chapters, we present two major extensions to our basic designs: an extension for network virtualization, and integration with other architectures in GENI.

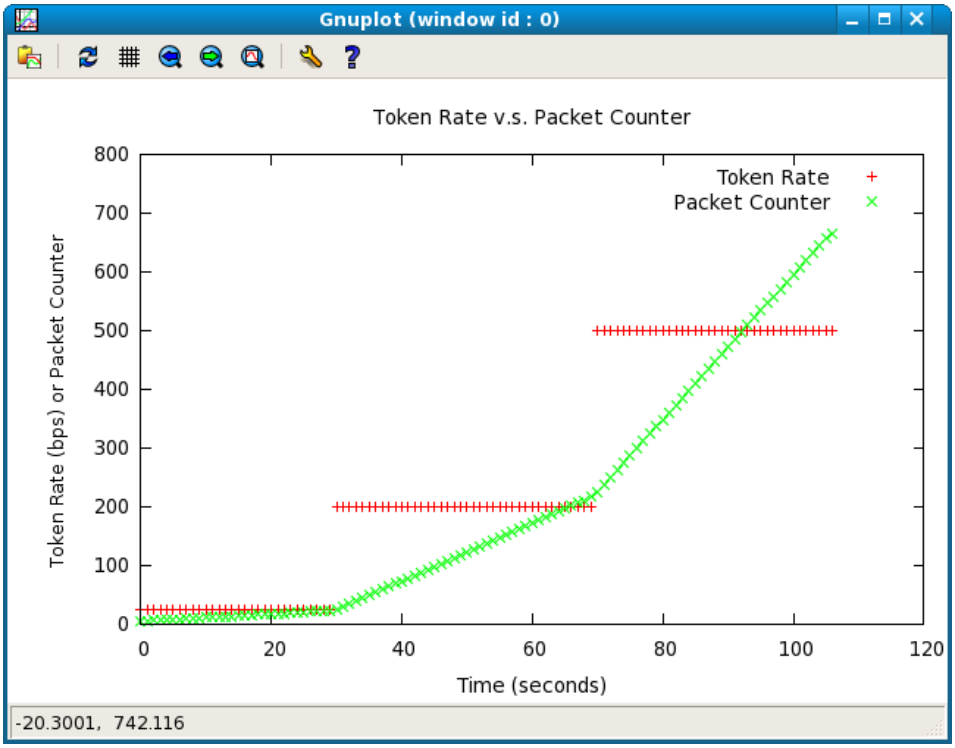

Figure 5.4: NSF SILO Demo Plot

Chapter 6

Understanding Network Virtualization

In recent developments in network research and technology, network virtualization has become an important trend. We believe network virtualization is critical to the future Internet, and any newly developed network architecture should be able to gracefully support network virtualization. In order to extend our SILO architecture to accommodate network virtualization, we deeply investigate network virtualization and present our definition and findings regarding this topic in this chapter.

6.1 Prosperity of Network Virtualization

Network virtualization has become a popular topic in recent years, and it is often mentioned in technical magazines, network device providers' white papers, textbooks and the academic community's research papers, etc.

Some survey papers [42, 43] attempt to give an overview of network virtualization, mainly from the point of view of the research community. However, they only address some aspects of network virtualization. Some Web sites, such as `networkvirtualization.com`, have become promoters and advocates of network virtualization. However, their view does not cover the entire spectrum of network virtualization either. With the rapid adaption of this term in different contexts by referring to different things, the term, network virtualization, bears much meaning.

This chapter discusses why people define network virtualization differently and reveals a comprehensive picture instead of just one aspect. We aim to give clear answers to fundamental questions like: "**What is network virtualization?**" and "**What are we virtualizing?**"

The rest of this chapter is organized as follows. First, in Section 6.2, we review network

virtualization-related technologies, which are mature and used in the industry. Then, in Section 6.3, we discuss network virtualization projects, proposals, and prototypes mainly driven by the research community. Then, in Section 6.4, we come back to the definition and answer the above two questions in after developing a deep understanding of network virtualization. Next, in Section 6.5, we discuss the trends of network virtualization and the challenges that we currently face.

6.2 Network Virtualization in the Industry

In this section, we examine the most popular network virtualization-related technologies that are already in commercial use. We elucidate how these technologies are related and how they evolve together.

6.2.1 Network Device Virtualization

In this section, we describe virtualization technologies of the fundamental building block of a network, such as Network Interface Card (NIC) and router. NICs generally reside on the edge, while routers consist of the intermediate nodes of the network.

NIC Virtualization

VMWare, Microsoft, and Xen [44] are among the most popular operating system virtualization solutions. One of the tasks of OS Platform Virtualization is virtualizing the network interface card (NIC), which enables the sharing of NIC hardware and virtual NICs (vNIC) for instances of virtual OS.

Figure 6.1 shows a general architecture of NIC virtualization, although the implementations from different vendors may vary. The cornerstone of NIC virtualization is vNIC, which is a software emulation of a physical NIC. There could be a dedicated IP address and a MAC address for every vNIC.

vNIC client can be any client who wants to use a vNIC. The most common vNIC client is a virtual OS, which can be called a virtual machine in VMWare, or a domain in Xen. Another common vNIC client is an operating system-level virtualization instance, such as a Solaris Zone.

Virtual Switch (vSwitch) is also a software emulation of physical switch. However, it might not support all features of a physical switch due to the nature of its usage [45]. vSwitch provides a traffic switch between vNICs and bridges vNICs with physical NIC(s); it performs functions such as traffic switch, multiplexer, scheduler, etc.

The links between vNIC and vSwitch are software-emulated links (not the virtual link we describe in Section 6.2.2). The bandwidth of this emulated link is only limited by the processing

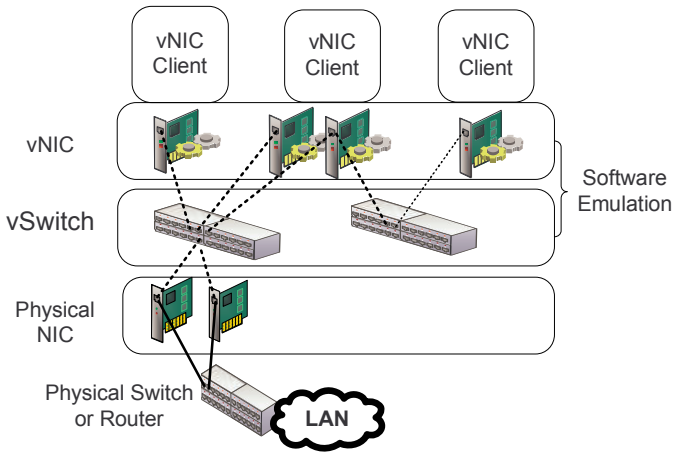

Figure 6.1: General NIC Virtualization Architecture

capabilities of the host itself. It is easy to set an upper speed limit of the emulated links to maintain the overall balance of traffic. However, most NIC virtualizaton implementations do not support guaranteed bandwidth. This is what makes the Sun Crossbow project [46, 47] distinct. Crossbow not only supports vNIC to reserve the hardware traffic lane and provides guaranteed bandwidth feature, but also offers a more elegant framework to manage resources. All of the above emulations happen either in the hypervisor or the host OS, in which Solaris Zone resides.

In Figure 6.1, if vNIC clients are servers, such as Web servers, DNS servers or firewalls, NIC virtualization actually provides a virtual network composed of virtual servers, virtual NICs, virtual switches, and virtual links. This is often called "network-in-box." In this context, the virtual network is actually a software-emulated network. It generates traffic that is injected to the real world through the only non-virtual/non-emulated physical NIC.

With increasing amounts of server virtualization in data centers, the traditional assumptions of server immobility and per-port-per-server have become invalid. Alternative ways to abstract vSwitch are used to solve this problem. In Figure 6.1, vSwitch can be abstracted to a more capable emulated switch, such as Cisco Nexus 1000v, which is embedded in the hypervisor. Another way to do this is by migrating vSwitch functionalities into the physical switch in Figure 6.1. The remaining software emulation merely acts as a multiplexer, and the physical switch is enhanced to be aware of vNIC. A detailed discussion of these alternative ways is out of the scope of this thesis. Please refer to [48] for more details.

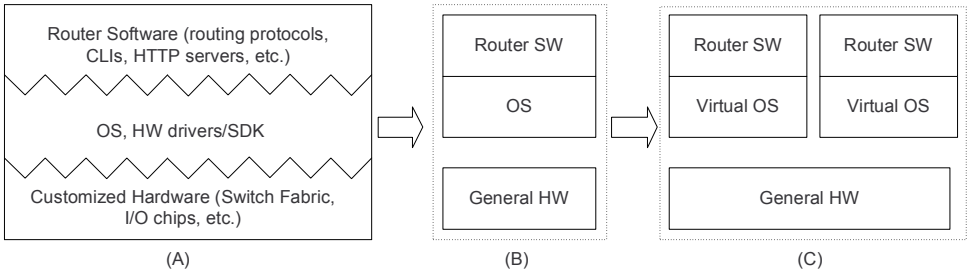

Figure 6.2: Routers in Virtual OS

Router Virtualization

In this section, we use router as a general term to refer to a network device such as a router or switch. Every router virtualization technology described in the following three subsections presents fundamentally different virtual routers (which is why they are called *virtual*).

Routers in Virtual OS Some popular operating systems used for routers are Cisco's Internetwork Operating System (ISO), VxWorks, Linux, and BSD. Figure 6.2-A shows a typical router whose software and hardware are highly coupled together. OS is often customized to the specially designed hardware for better performance. Router software, which includes routing protocols, runs in these customized OS.

There have been continuous efforts to separate software from hardware (Figure 6.2-B), especially in some Linux distributions. A list of router software distributions is provided in [49]. Most of those distributions can run on standard X86 hardware architecture, and some of them can even be installed in virtual OS, like VMWare or Xen (Figure 6.2-C). Vyatta [50] is one example where generalized router software is provided, and it directly competes with Cisco and other router manufactures in the low-end router market.

Those routers in virtual OS can of course be called virtual routers. In this context, the term *virtual* means these routers share hardware resources, such as CPU, memory, hard drive, network card, etc., with other instances of virtual OS in the same physical machine. These virtual routers are vNIC clients in Figure 6.1. They reinforce the concept of "network-in-box" described in Section 6.2.1.

Router Control-plane Virtualization The core of routers consists of routing tables, which lead packets to destinations. A router may have only one routing table for all packets, and the routing table is maintained by a single process. Some routers could have multiple routing tables, and each routing table serves a VLAN. These multiple routing tables could be maintained by a single process or by multiple processes (one process for each routing table). This technology

is named Virtual Routing and Forwarding (VRF). VRF is a virtual routing instance, but it is not a fully virtualized device yet.

Other advanced routers support the virtual routing instance, which not only have separate routing tables, but also have (logically) separate routing protocols, configurations, etc. They are different from routers in virtual OS, as discussed in Section 6.2.1. There is still one OS in the router, and virtual routing instances are logically separated, and they share OS and hardware. Various names have been given to virtual routing instances, such as a virtual router, logical router, and context, etc.

Hardware-partitioned Router Some routers support hardware partition and host multiple routing instances in a single device. They are called "protected system domains" [51] by Juniper Networks, or "logical router" by Cisco Systems. The motivations for these hardware-partitioned routers are mainly saving spaces and power in Points of Presence (PoP) of network carriers, and reducing management cost. Hardware is typically partitioned per line card, so that it often can be viewed as a router per line card.

6.2.2 Link Virtualization

Link Virtualization provides virtual links. However, both Link Virtualization and virtual link might refer to distinct concepts in different contexts. We review those different interpretations and reveal the layers of link virtualization.

Physical Channel Multiplexing

There are various ways to multiplex communication channels over a physical media. Two most popular multiplexing technologies are time-division multiplexing and frequency-division multiplexing.

When discussing link virtualization, what is the link? This is the first question we need to clarify. If the link is a physical media like a fiber, link virtualization might be identical to multiplexing.

Generally multiplexing is generally not considered to be link virtualization. However, we would like to point out that multiplexing bears the exact same idea as virtualization. The physical medium is split to channels, and the sender and receiver have the illusion that they own the physical medium. For similar reason, virtualization might be redeemed as a high level of multiplexing.

Manipulating Channels

In this subsection, we refer to links as channels that cannot (would not) be split by multiplexing. Link virtualization might be used to refer to the means by which we put these channels together to form virtual links.

Virtual Circuit In a traditional telephone call, a virtual circuit is always established for every call. A channel is essentially a time slot. A virtual circuit is a series of time slots between two ends. The conversation is carried on these time slots, and the users have the illusion that they use the entire circuit/telephone line.

In this case, channels are assigned along the way to form a virtual circuit, which is a virtual link in this context.

Optical Bandwidth Virtualization With advances in optical technologies, we are able to combine or split bandwidth/wavelength in a manageable manner. The basic bandwidth unit could be called the sub-rate/sub-wavelength/substrate/channel.

Let the assumption be made that we have a basic unit with 10G bandwidth. In order to provision a 40G bandwidth, we may combine 4 10G units. These units could be from one physical channel or different channels. When we need 100G bandwidth, we could reuse the previous 4 10G units and combine additional 6 10G units. Furthermore, we may use this practice to provision a whole network with guaranteed 100G bandwidth. The link can be called a virtual link, and the network can be called a virtual network from the perspective that its bandwidth is provisioned using the above technology. Optical bandwidth virtualization also can be used to provision L1VPN, which we will describe in Section 6.2.3.

Infinera is one of the pioneers in providing a bandwidth virtualization solution [52]. In the traditional optical network, the services are highly coupled with particular physical devices. Bandwidth virtualization enables this decoupling, and the services are not bound to particular devices, but to virtualized bandwidth.

In this case, channels are combined to provide higher bandwidth channels, which are virtual links in this context.

Link virtualization , as discussed thus far, is highly dependant on the physical properties of links. We cannot achieve these if links' physical properties do not support it.

Data Path

In this subsection, we refer to a link as a data path. This might be the most common understanding of link virtualization. In this case, a virtual link equals a virtual data path. What we manipulate are not channels themselves, but the data inside these channels.

Table 6.1: Comparisons of Link Virtualization

Names	**Enabled by**	**Examples**
Physical Channel Multiplexing	links' physical properties	FDM, TDM
Manipulating Channels	links' physical properties and nodes	Virtual Circuit in PSTN, Optical Bandwidth Virtualization
Data Path	nodes	VLAN, MPLS, GRE tunnels

A virtual data path does not depend on links' physical properties. It is actually provisioned by nodes. Nodes use various technologies to direct data flows through links and to form virtual data pathes. We will discuss popular technologies for virtual data paths in the following two sections.

Tags and Labels 802.1q VLAN tags enable different VLANs to share physical media and to be logically separated. VLAN tags are more about sharing and can be used to distinguish data from different VLANs. At the same time, they are also employed to help form data paths for the broadcasting domain. We will discuss VLAN in greater detail in Section 6.2.3.

Labels and tags are essentially the same. They are certain fields in a packet designed to serve certain purposes. The labels in ATM, Frame Relay and MPLS/GMPLS are used to determine the data paths. These data paths are called virtual circuits as well, because they provide certain circuit advantages but not a real circuit. Please note the above virtual circuits are different from what we mentioned in Section 6.2.2.

Nodes are also aware of those tags and labels, so that they may point traffic in the correct directions. This is a very important concept. It basically shows that the virtual data path is not provided by links, but by nodes.

Labels and tags do not partition any bandwidth on links, and they are merely identifications and sharing mechanisms.

Tunnels and Encapsulations Nodes in a network are not always directly physically connected. Tunnels (that often use encapsulation) provide virtual links (or logical links) to connect network devices. For example, some protocols of network devices have the illusion that this device has a direct connection to other devices through tunnels. Some popular technologies are Generic Routing Encapsulation (GRE) tunnels, IP Sec tunnels and MPLS LSP tunnels, etc. Again, these technologies do not partition any resource like Bandwidth Virtualization. They are essentially overlay links and the fundamentals of overlay networks, which are discussed in Section 6.2.3.

Table 6.1 summarizes and compares link virtualization, as discussed above.

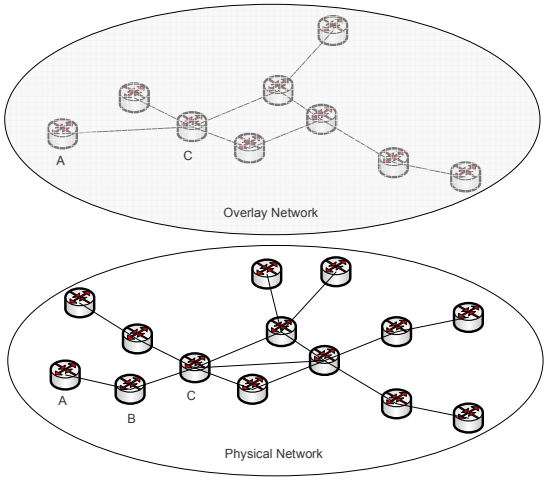

Figure 6.3: Overlay Network

6.2.3 Virtual Networks

A network is called *virtual* for different reasons. So far, we have already mentioned two types of virtual networks: "network-in-box" in Section 6.2.1 and "bandwidth-virtualized network" in Section 6.2.2. In this section, we review several other types of networks that also have been claimed as virtual networks. By comparing their fundamental different emphases, we answer why each of them is called a virtual network from a different perspective.

Overlay Network

An overlay network is a network built upon an existing network by various technologies, but mainly tunneling and encapsulation, in order to implement new network services.

A great benefit of overlay networks is implementing new network services economically by making use of existing networks. In the physical network in Figure 6.3, Nodes A, B and C are connected by two links: Link AB and Link BC. We need implement a new network service, such as a new routing protocol, between Node A and C. We can use a tunnel to connect Node A and C. Only Node A and C need to be modified to be aware of this new service; Node B remains as it is. Node B still forward data between Node A and C, but is not aware of the new service. The new service on Node A and C give these two nodes the impression that they are connected to each other by Link AC without knowing the existence of Node B.

There are several very well-known overlay networks that users encounter in daily life. We compare these most common overlay Internet access networks in Table 6.2. In this table, PSTN stands for Public Switched Telephone Network; DSLAM means Digital Subscriber Line Access Multiplexer; and CMTS refers to Cable Modem Termination System. For every overlay

Table 6.2: Overlay Internet Access Network

Names	**Overlaid upon**	**Starts from**	**Ends at**
dial-up	PSTN	dial-up modem	ISP termination modem
DSL	PSTN	DSL modem	DSLAM
cable	cable network	cable modem	CMTS

Figure 6.4: Virtual Private Network

network, we show that what network it overlaid upon, where it starts (Node A in Figure 6.3) and where it ends (Node C in Figure 6.3). As the networks are bidirectional, the nodes they start are also the nodes they end; and we take the users' point of view in describing starting and ending in Table 6.2.

In overlay networks, network topologies are virtualized. All new service-aware nodes form a virtual network. Overlay is also a key technology for implementing a virtual private network and a virtual sharing network, as described in the following two sections.

Virtual Private Network

A Virtual Private Network (VPN) is an assembly of private networks connected and isolated from public network, such as the Internet. An organization spreading over geographically distant locations needs VPN to connect its offices. Employees who work from home require VPN to access the company's internal network. As Figure 6.4 shows, VPN emphasizes on connecting different private networks.

Layers 2 and 3 VPN are mature technologies. For Layer 2 VPN, the VPN provider's network is virtualized as a Layer 2 switch. Customers' sites are able to build their own routing infrastructures. Similarly for Layer 3 VPN, the VPN provider's network is virtualized as a layer 3 router.

Since 2005, Layer 1 VPN has been undergoing a rapid process of standardization. The fundamental difference between L2/3VPN and L1VPN is the networks on which they are supposed to operate. L2/3VPN assumes an IP/MPLS+BGP core, while L1VPN is mainly designed to run upon time-division multiplexing (TDM) networks, such as SONET/SDH, and WDM optical networks. In L1VPN, customers (the edge routers of private networks) are able to request Layer 1 data paths in the VPN providers' network, and then customer sites are connected by

Table 6.3: VPN Summary

Type	**Provider's Entire Network Virtualized as**	**Mainly Designed for**
Layer 3	router	IP/MPLS+BGP core
Layer 2	switch	IP/MPLS+BGP core
Layer 1	Layer 1 connection	TDM & optical network

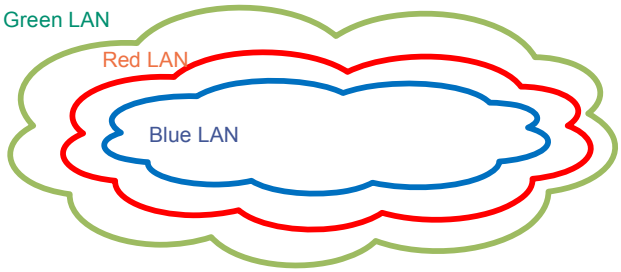

Figure 6.5: Virtual LAN

these Layer 1 data paths. However, L1VPN is provisioned by higher layer provisioning protocols, i.e., the requested data path is determined and established by protocols such as GMPLS. L1VPN is still fairly new and under development. Please refer to L1VPN RFCs [53] and other drafts for more detailed information.

Table 6.3 summarizes three types of VPN by pointing out what VPN provider's network (or the Internet) is virtualized as.

In VPN, network topology is also virtualized. All private networks form a virtual network, which is isolated from the provider's network.

Virtual Sharing Network

First of all, we need to point out that Virtual Sharing Network is not an official name for the network we describe in this section. A more popular name for it is simply virtual network[54]. However, we call it Virtual Sharing Network (VSN) in this thesis in order to differentiate it from the Overlay Network and VPN.

Virtual LAN (VLAN) is the most common and easily understandable form of VSN. In Figure 6.5, the green LAN, the red LAN and the blue LAN share the same physical LAN infrastructure, and at the same time, they are segmented within the boundary of broadcast domains. Sharing and segmentation are two main features of VSN.

General VSN is an extension of the VLAN concept into a broader network. Figure 6.6 shows a typical scenario of VSN within a large or medium-sized corporate network. The guest network, employee network and administrator network might share the same access point (wireless hot spots, LAN access), physical switches and routers, and some severs, etc. However, these

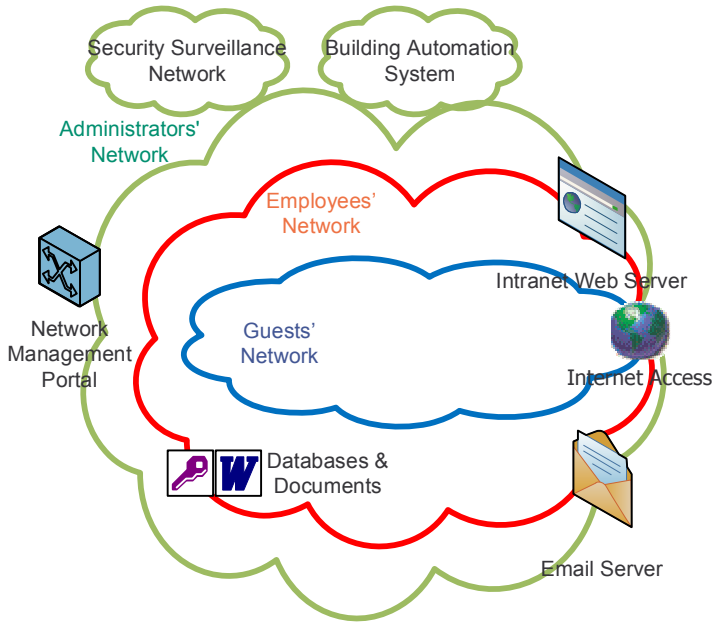

Figure 6.6: Virtual Sharing Network

networks are also properly segmented as virtual networks. All virtual networks are able to access the Internet; the employee network and administrator network can access email servers, Intranet Web servers, etc.; the employee network can access documents and databases; the administrator network provides the ability to configure physical devices, and even access to other networks like security surveillance networks and build automation systems.

Instead of building a separate physical network for guests, employees and administrators, provisioning a virtual network for each user group is a better solution in terms of cost, efficiency, and so on. The basic requirements of these virtual networks are still sharing the same physical infrastructure and being properly segmented.

VSN is usually extended by the VPNs discussed in Section 6.2.3, i.e., a virtual network in VSN could be extended to multiple physical locations. At this point, the term *virtual* has two meanings. It is a VSN virtual network because it shares the same physical network with other VSN virtual networks. It is a VPN virtual network because it forms a virtual network with peer networks located in other physical locations. Figure 6.7 explains the above extension.

Various technologies can be used to build VSNs. Figure 6.8 shows the most common technologies as building blocks of VSN. These include access control, VLAN, VPN, data-path virtualization, control-plane virtualization, etc. Depending on the design of VSN, a different set of technologies might be used. Designing VSN is an art and often requires experience and following best practices.

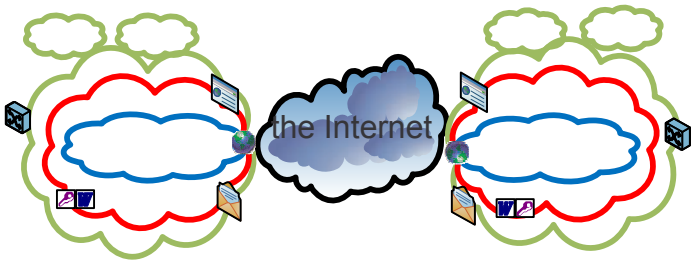

Figure 6.7: Extended VSN

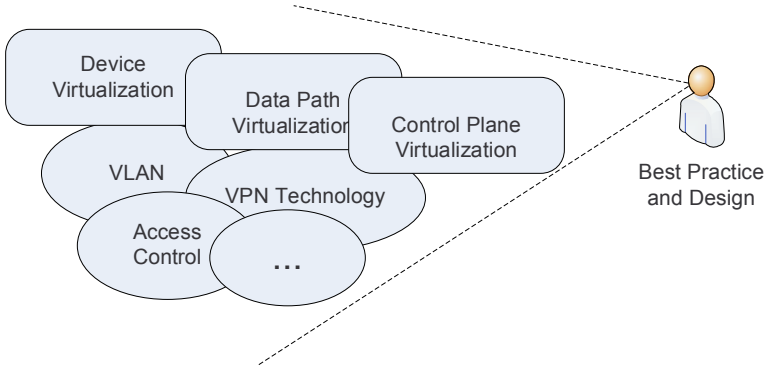

Figure 6.8: Technology Building Blocks of VSN

The VSN solution (called "network virtualization" by Cisco) has been promoted as one of Cisco's important business strategies. Its business benefits are addressed in [55].

Relations and Comparisons

In Figure 6.4, the VPN running upon the VPN provider's network is usually an overlay network, but not always. VPN does not necessarily imply that overlay is used. The legacy VPN technology that does not use overlay is called peer-to-peer VPN [56, 57].

Figure 6.9 shows the relations between the overlay network, VPN and VSN. Table 6.4 gives some examples that fit the different areas (from A to G) in Figure 6.9.

As Figure 6.9 and Table 6.4 show, the overlay network, VPN and VSN often overlap and intertwine with each other in real deployment.

In those networks, physical topologies have been presented as different logical topologies. Table 6.5 compares their emphasis and what forms a virtual network, i.e., it explains the fundamental different reasons why they are called virtual networks.

There are other forms of virtual networks from a different perspective, especially in the academic community. We will discuss these in Section 6.3.

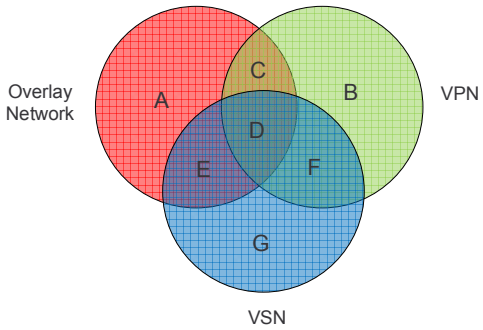

Figure 6.9: Relations of Overlay Network, VPN and VSN

Table 6.4: Examples of Overlay Network, VPN and VSN

Areas in Figure 6.9	**Examples**
A	Dial-up, DSL, Cable Internet
B	peer-to-peer VPN [56, 57]
C	Most modern VPNs
D	VPN in Extended VSN (Figure 6.7)
E	peer-to-peer VPN in VSN
F	Overlay used in VSN
G	VLAN

6.3 Network Virtualization from the Perspective of the Academic Community

In this section, we focus on projects that are mainly driven by the academic community and explore the new meanings of network virtualization introduced by these projects.

We categorize selected projects based on their primary approaches and discuss them in detail in the following subsections, and continue to explore how the academic community answers the question: "**what are we virtualizing?**" For full survey papers, please refer to [42, 43].

6.3.1 Testbeds Provisioned by Network Virtualization

The testbeds discussed in this section, use network virtualization to aid experimenters (users of testbeds) in sharing the same physical testbed and keeping each other isolated. We focus on what are virtualized and provided to the experimenters.

PlanetLab

PlanetLab is a research testbed, which was established in 2002 by the joint efforts of Princeton, Intel, UC Berkeley and the University of Washington. As of November 2009, there were 1,045

Table 6.5: Virtual Networks Summary

Type	**Emphasis**	**What Forms a Virtual Network**
Overlay Network	being made aware	new service-aware nodes
VPN	connecting	private networks
VSN	sharing	part of the physical infrastructure

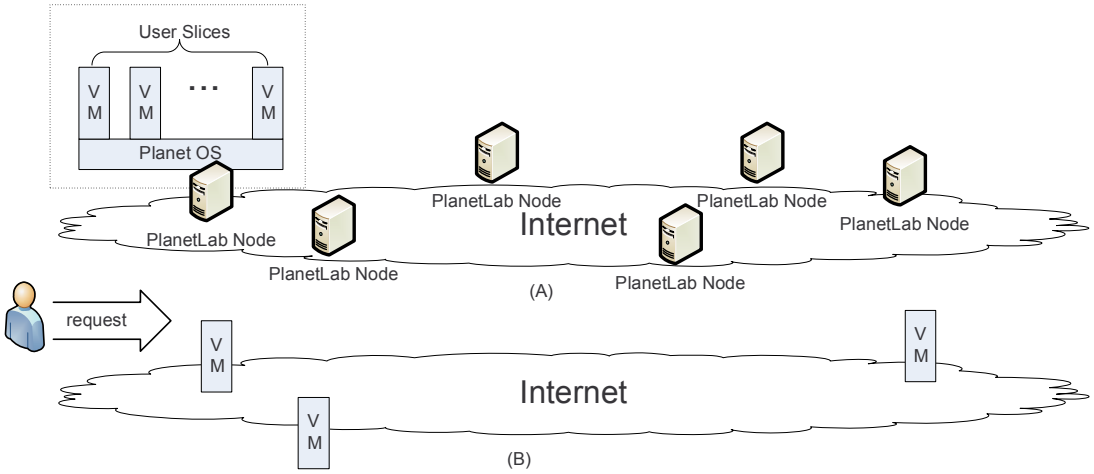

Figure 6.10: PlanetLab

PlanetNodes available at 490 sites across the world, mainly in the United States, Europe, China and Japan.

As Figure 6.10-A shows, PlanetLab is composed of PlanetNodes distributed around the world. PlanetNodes are dedicated servers running PlanetOS (customized Linux), and they are able to spawn Virtual Machine (VM) slices as requested. There are no dedicated links between the PlanetNodes, and the communications between PlanetNodes occur through the Internet, i.e., PlanetLab is overlaid upon the Internet. So the performance of a particular experiment running on PlanetLab is dependent on the Internet traffic. We would also like to highlight that there is no presumed topology for these PlanetNodes.

Figures 6.10-B shows that with the help of the PlanetLab management system, users are able to pick some or all nodes, and to request a virtual machine slice from each node. As PlanetNodes are geographically distributed around the world, the real world scale testbed can be requested and established. Users can utilize the PlanetLab APIs to develop customized applications for their experiments and to distribute them on a global scale. It may also be said that PlanetLab provides a user-defined overlay network. What network topologies the users want to deploy, and what the services the customized applications want to provide truly depend on the users' design.

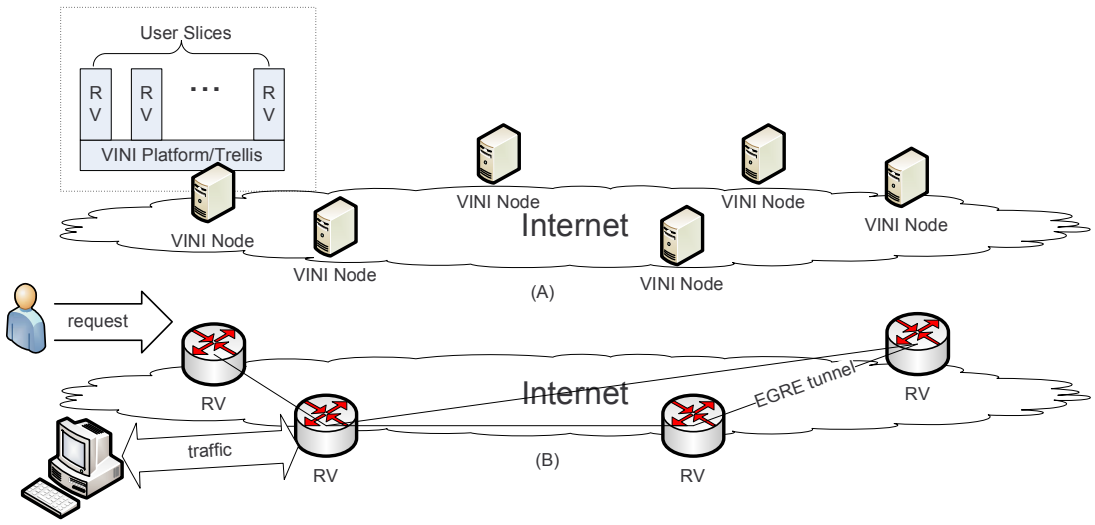

Figure 6.11: VINI

The design initiatives and blueprint are described in [58]; the details of PlanetOS are in [59]; Linux VServer [60] is used to provide virtual machine instances on PlanetNode; and the security aspect of PlanetLab is presented in [61].

VINI & Trellis

A Virtual Network Infrastructure (VINI) is an extension of PlanetLab from a general purpose testbed with distributed resources to a specific enhanced network protocol and service testbed.

As Figure 6.11 shows, VINI is also an overlay on the Internet, but it has three fundamental differences from PlanetLab. (1) Every slice (also called a virtual node) is enhanced as a router, not just as a general purpose slice. This concept of routers in virtual OS has been discussed in Section 6.2.1. (2) Tools are provided to establish tunneling connections between virtual nodes. (3) Traffic can be directed between VINI and the outside world to simulate realistic conditions.

VINI was initially implemented upon PlanetLab as PL-VINI [62]. Later, it was upgraded to its own physical testbed with a new software platform called Trellis [63]. As of November 2009, there were 42 nodes at 26 sites.

Trellis uses the Ethernet GRE (EGRE) tunnel to simulate Layer 2 links between virtual nodes. Although there might be a dedicated uplink for the physical VINI nodes, these tunnel links do not provide guaranteed bandwidth because they are overlaid on the Internet. In order to enhance the performance of bridging virtual NICs (vNICs) of "routers in virtual OS" to the EGRE tunnels, the standard Linux bridge (similar to *vSwitch* in Section 6.2.1) has been optimized as *shortbridge* [63].

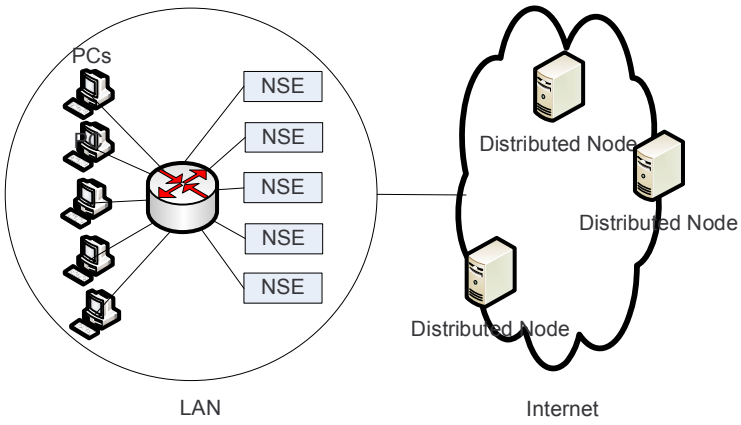

Figure 6.12: Emulab

Emulab

Emulab [64, 65] is a research network testbed developed at the University of Utah. Since the setup of late 1990s, Emulab has been an important platform for networking reach and teaching. It was promoted as NetBed for a short period of time during the first few years of its inception.

As its name indicates, Emulab mainly emulates networks and returns them to users. Figure 6.12 shows the main components of Emulab. The right part of Figure 6.12 shows overlaid distributed nodes on the Internet, which work very similarly to PlanetLab and VINI. In addition, Emulab uses the left part of Figure 6.12 to actually emulate the network that users request. Within a physical site (presented by the circle in the picture), PCs are connected to a switch within a LAN level. When users request links with specific delays, a link from the LAN is assigned to emulate the requested link, which is implemented by inserting a Dummynet node [66, 67]. A Dummynet node is nothing more than a link emulator sitting between two PCs. Emulab also incorporates the NS Emulate (NSE) [68] to allow the purely software-simulated traffic to interact with the PC-cluster simulated network and overlaid distributed nodes.

In recent years, Emulab and PlanetLab have been part of the GENI (Global Environment for Network Innovations) initiative [13], and the interactions between those two major testbeds have been addressed and developed.

VIOLIN

Virtual Internetworking on OverLay INfrastructure (VIOLIN) [69, 70] was an early effort before VINI to build an overlay network testbed. It was implemented on PlanetLab and is essentially like PL-VINI described above. VILION software is implemented within a Linux UML container, which runs in a PlanetLab slice. A UML container is a virtual host, a virtual switch, or a virtual

router. VIOLIN extends UML to enable UDP tunnels between the UML containers. As VIOLIN is independent of PlanetLab, it is easy to be implanted on other testbeds, such as Emulab.

6.3.2 Network Management

X-Bone

The X-Bone [71, 72] is a management tool that eases the effort of automatic configuration and the management of overlay networks upon the IP network with enhanced security and monitoring. It can be viewed as a tool to deploy overlay networks across the IP network. It also provides an isolation and resource allocation mechanism between multiple overlay networks. The recent development and road map are well presented on the X-Bone Web site [73].

UCLPv2

User Controlled LightPaths version 2 (UCLPv2) [74] is a set of software in the form of Web services, which provides user-friendly, unified interfaces to various network elements (optical switches, routers, etc.). The Web services are said to simplify the process of combining or partitioning network resources such as bandwidth. LightPaths in this context not only refer to wavelengths, but also to SONET circuits, MPLS label switch paths (LSP).

UCLPv2 itself does not virtualize any network resource, but merely provides an abstraction of network elements. A network provisioned by UCLPv2 is called Articulated Private Net (APN). Although it is claimed to be the next generation VPN [74], it does not extend the concept of VPN itself.

6.3.3 Architecture Aspects

A Virtual Internet Architecture

A Virtual Internet Architecture [75] is an overlay network architecture built upon a base net (the Internet). The authors point out that Virtual Internet (VI) is in a very preliminary stage, and they use computer memory as an analogy for analysis. Just as virtual memory covers all the details of physical memory and looks like it is owned by an OS process, VI should also cover the details of the physical network and give VI users an abstraction of the network. VI is designed to support recursion and revisitation, which provide maximum flexibility. "Recursion allows VIs to be stacked, providing new opportunities for fault tolerance and path diversity. Revisitation allows a single base network node to emulate multiple VI nodes, allowing a VI to emulate larger networks than the base on which they are deployed" [75].

This architecture describes the host, router, link and topologies as network resources, and it examines the impact of using a virtualized environment over network resources.

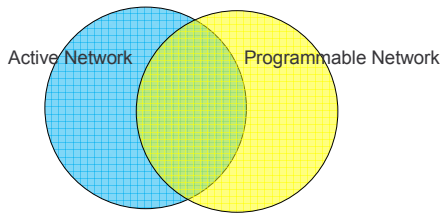

Figure 6.13: Relations between Active Network and Programmable Network

Active Network and Programmable Network

The traditional network's computation mainly resides on end systems; the routers/switches simply forward packets to destinations, i.e., the network works passively. In the Active Network [76, 77], there are not only data in the packet, but also programs. Programs can be executed by routers/switches to dynamically modify the operation of the network.

One way to implement an Active Network is by programmable hardware. A network composed of programmable hardware is called a programmable network. The relation between Active Network and Programmable Network is shown in Figure 6.13.

Although Active Network or Programmable Network does not necessarily indicate network virtualization, both of the two technologies can be used to implement network virtualization.

A detailed description of Active Network and Programmable Network is beyond the scope of this thesis, and please refer to [78, 79, 80, 81, 82, 83].

6.4 Back to the Definition

6.4.1 Others' Interpretations

Given the above discussion, network virtualization is such a broad term that it has been used to refer to various concepts. We will review how others interpret network virtualization in this section.

"In computing, *network virtualization* is the process of combining hardware and software network resources and network functionality into a single, software-based administrative entity, a virtual network. Network virtualization involves platform virtualization, often combined with resource virtualization"[84]. This definition points out that virtual network is the goal and result of network virtualization and indicates that resource virtualization is involved.

"The term *network virtualization* describes the ability to refer to network resources logically rather than having to refer to specific physical network devices, configurations, or collections of related machines. There are different levels of network virtualization, ranging from single-machine, network-device virtualization that enables multiple virtual machine to share a single

physical-network resources, to enterprize-level concepts such as virtual private networks and enterprize-core and edge-routing techniques for creating subnetworks and segmenting existing networks"[85]. This definition also points out that network resources are virtualized and enumerates various *levels* of network virtualization, but the list of levels is incomplete, and the concept of *level* remains ambiguous.

"*Network virtualization* is a method of combining the available resources in a network by splitting up the available bandwidth into channels, each of which is independent from the others, and each of which can be assigned (or reassigned) to a particular server or device in real time. Each channel is independently secured"[86]. This merely defines the physical channel multiplexing discussed in Section 6.2.2. It does not cover the entire spectrum of network virtualization.

6.4.2 Some Comparisons

Before delving further into our definition of network virtualization, we present some ambiguous terms and compare them in this section.

Network Virtualization vs. Network Simulation and Network Emulation

Network Simulation refers to the means of simulating network behaviors by software and/or hardware without the actual network being present. OPNET [87] and NS2/NS3 [88, 89] have been among the most popular network simulation tools. Networks presented by network simulation tools are not part of the real network. They use discrete events to drive simulation and provide simulated networks. Although a "network-in-box" scenario is also presented in comparison to what was described in Section 6.2.1, they are fundamentally different in terms of being part of the real network.

There is a distinction between network simulation and network emulation. *Network emulation* tools usually refer to those software or hardware modules deployed in a real network or testbed to emulate certain properties of links (such as bandwidth, latency, etc.), or traffic patterns (packet lost, packet duplications, jitter, etc.). Dummynet [66, 67], Netem [90], and NistNET [91, 92] are some examples of popular network emulation tools.

As discussed in Section 6.3.1, in Emulab, NSE [68] relays the simulated world to the real world, and it enables simulated traffic to be injected in the real network and processes the traffic from the real network. In this case, the simulated network comprises part of the virtual network provided by Emulab.

Occasionally, network virtualization is used as an alternative term for network simulation or emulation [93]. Because an emulated network is not a real physical network, some people argue that it is a virtual network and refer to emulation as network virtualization. This once

again proves that network virtualization is such a broad term that can be interpreted in various ways. However, we do not encourage this usage, considering (1) network simulation and network emulation are better terms to use; and (2) network virtualization has already been used quite broadly.

Virtualization, OS Platform Virtualization and Network Virtualization

In Computer Science, *virtualization* is a general term used to refer to the abstraction of computing resources. Virtualization presents various forms of virtualized resources and gives users the illusion of sole ownership. The key to virtualization is virtualizing fundamental resources.

What are the fundamental resources of an OS platform? In OS Platform Virtualization, computer hardware components are virtualized, such as CPU, memory, hard drive, network card, etc. These components are viewed as fundamental resources in OS Platform Virtualization.

What are the fundamental resources of a network? A relatively easy question for OS platform virtualization becomes a little harder and trickier when pertaining to network virtualization. As the fundamental components of a network are nodes and links (vertices and edges in graph theory), it is easy to conclude that nodes and links are network resources. Node virtualization is discussed in Section 6.2.1, and link virtualization is discussed in Section 6.2.2.

The real difficulty in defining network virtualization is dependent on the following question. "Are these two obvious resources the ONLY resources?"

Argument about Topology Topology defines how nodes are linked together. Some might argue that topology is a new type of resource, as links can be added or deleted to form a virtual topology. This is an arguable position in terms of topology existing as a being. However, virtual topology cannot be achieved without any form of node or link virtualization. Topology is still built upon nodes and links. In this sense, it is not a fundamental resource and can be viewed as a derived resource.

Addresses When we have a network composed of nodes and links, we must have a mechanism to address/name nodes or links in order to make use of the whole network. Once we decide to use a certain methodology to address nodes, the addressing itself becomes an essential resource. It can be virtualized with the aid of various ways of address translation. When the number of nodes and links is larger than the address space, nodes and links compete for the limited addresses. We view addresses as a fundamental resource as well.

Figure 6.14 shows the network resources we have identified. For every particular type of network virtualization, the virtualized resources might vary significantly. We review this in detail in Section 6.4.4.

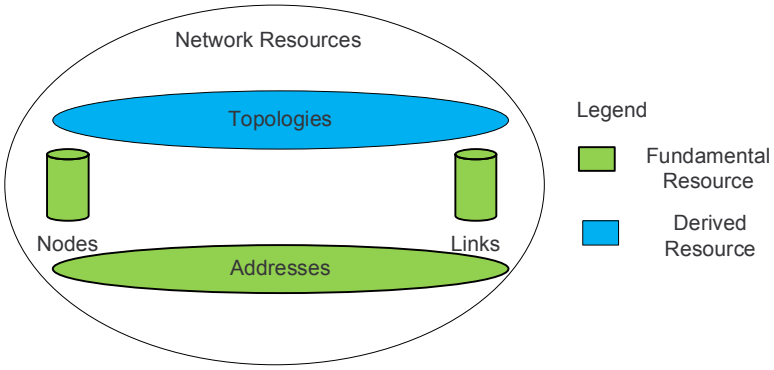

Figure 6.14: Network Resources

Virtual, Virtualize, Virtualized and Virtualization

The primary meaning of *virtual* given in answers.com is: "existing or resulting in essence or effect though not in actual fact, form, or name." This term focuses on giving an illusion and does not necessarily indicate that resources have been virtualized. On the other hand, *virtualize*, *virtualized* and *virtualization* indicate resources have been virtualized. There is a subtle difference between these words.

6.4.3 What Are We Virtualizing?

In this section, we first summarize node virtualization and link virtualization in terms of their virtualized resources in Table 6.6. Then we review all virtual networks discussed in this chapter and point out their virtualized resources in Table 6.7. Selected academic projects are listed in Table 6.8 from the same perspective as well.

6.4.4 Our Definition

From the above discussion, we conclude the best way to define network virtualization is focusing on how it is related to network resources. We define network virtualization as follows: **Network virtualization is any form of partitioning or combining a set of network resources, and presenting it to users such that each user can use the partitioned or combined resources and have a unique separate view of the network. Resources can be fundamental (nodes, links and addresses) or derived (topologies).**

Table 6.6: Summary of Node and Link Virtualization

Category	Sub-category		Virtualized Resources
Node	NIC virtualization	techniques in Xen and VMWare	NIC is virtualized as vNIC with full interfaces to OS
		techniques in Crossbow	vNIC with guaranteed bandwidth
	host/endpoint		same as OS virtualization
	router virtualization	routers in virtual OS	same as OS virtualization
		router control plane virtualization	routing table, routing protocols and configurations, etc.
		hardware-partitioned router	Power and chassis are shared. No resource is virtualized.
Link	Bandwidth Virtualization		Bandwidth
	Tags and Labels		Data are identified by tags.
	Tunnels		Data are encapsulated.

Table 6.7: Summary of Virtual Networks

Type	Virtualized Resources
Network-in-box	The network is emulated by software; NICs are virtualized.
Bandwidth-virtualized network	bandwidth
Overlay	Links are virtualized by tunnels; addresses might be virtualized.
VPN	Links are virtualized by tunnels; addresses might be virtualized.
VSN	links, nodes, and addresses

6.5 The Trends of Network Virtualization

6.5.1 Network Leasing

With more mature network virtualization technologies, network owners are able to lease part of their networks to others by means of virtual networks. Although VPNs (discussed in Section 6.2.3) have served this purpose a long time, the new approach [94] provides a broader impact by decomposing the role of Service Provider.

Traditional Internet Service Providers, such as AT&T and Time Warner, own physical infrastructures and provide services, such as Internet Access, corporation VPN, etc. Network virtualization allows the owners of physical infrastructures to provision virtualized infrastructures and to act as infrastructure providers. Of course, an infrastructure provider can be a service provider at the same time and can provide services upon its own infrastructure. However, it is also able to lease unused resources in the form of virtualized infrastructure to other

Table 6.8: Summary of Selected Academic Projects

Name	**Virtualized Resources**
PlanetLab, VINI, Emulab	Links are virtualized by tunnels; nodes are virtualized by OS virtualization.
X-Bone	same as overlay network
UCLPv2	bandwidth
Virtual Internet Architecture	host, router, and link

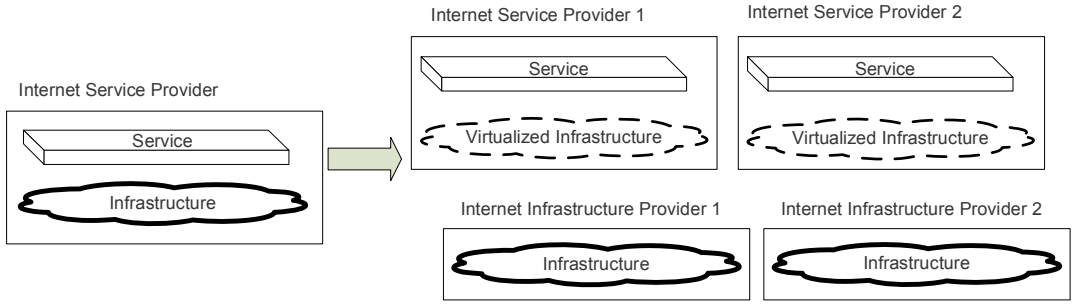

Figure 6.15: Decomposition of an Internet Service Provider

service providers. A service provider can lease from one infrastructure provider or from several infrastructures. Figure 6.15 shows the decomposition process.

6.5.2 Testbeds for Next Generation Internet

Some researchers have argued that the current Internet is ossified [95] and needs to be improved. For practical reasons, redesigning the Internet from the ground up and not considering the current existing infrastructure (in a clean-slate manner) are less appreciated in comparison to improving the current Internet [34]. Network virtualization is a natural solution to provide virtualized infrastructure as testbeds to experiment with newly designed architecture, and this is a smooth way to overcome the Internet impasse [96, 97].

Global Environment for Network Innovations (GENI) [13] in the United States and 4WARD [98] in the European Union are the two biggest Next Generation Internet initiatives. Both take network virtualization as one of their most important strategies.

GENI is a collection of research projects, most of which are clustered around four control frameworks, PlanetLab, ProtoGENI (descendent of Emulab), ORCA (Open Resource Control Architecture) [99, 100] and ORBIT [101]. Broadly speaking, GENI does not aim to provide the Next Generation Internet; it hopes to provides researchers ways to explore all the possibilities and addresses most of the challenges in the network virtualization area. GENI aims to design

a streamlined process to instantiate virtual networks, to allow on-demand addition of resources to these networks and to manage the virtual networks.

4WARD is a project funded by the European Union that has similar goals as GENI. With regard to virtualization, the specific goals of 4WARD are to provide: virtualization of network resources, virtual networks and virtualization management. Network resources to be virtualized could include servers and links. 4WARD focuses on a generalized approach to allow virtualization of different resources that form a unified virtualization framework. Both wireline and wireless resource virtualization are supported in 4WARD.

6.5.3 Research Challenges

While there are many challenges, we address the two main challenges of network virtualization in this section.

Control Frameworks

Control frameworks are the keys to enable network virtualization. Four control frameworks under GENI collaborate and compete with each other. We envision that multiple control frameworks would succeed and federate (work together) with each other to compose a future global control framework.

One aspect of control frameworks is resource control, which includes polling resources, leasing resources to requesters, interacting with other resource control frameworks and acting as a resource broker. This is crucial to network leasing, as discussed in Section 6.5.1. Another aspect of control frameworks is controlling resource to enable initialization, deployment, and monitoring, etc.

If a control framework is not open to the outside world, it makes the control itself easy while losing the power to federate with other control frameworks. On the other hand, if a control framework is too open, it is very flexible has the greater risk of being abused. There are fine balances and philosophies behind these control frameworks.

Security

In a virtualized network, security issues can be divided into three parts. Firstly, some issues are migrated directly from those of virtualized operating systems. For instance, in operating systems, the hypervisor should be well secured because it controls all the virtual machines and can potentially affect all the VMs. Similarly, in virtualized networks, the substrate that controls the different virtualized networks should be protected. Multiple users are accessing the same physical network resources, such as routers. Isolation between the different virtual networks is essential to maintain the illusion of virtualization. On the other hand, isolation

can only provide limited security and privacy with the use of current encryption techniques. As the physical router is still the same, it does not eliminate the risks involved with attacking the router itself. If a user in one virtualized network is somehow able to detect the presence of another virtualized networks, then the illusion of virtualization is broken.

Other security issues in virtualized networks are based on the goals of network security. These include well-known goals such as confidentiality, integrity and availability. Several solutions, such as authentication and intrusion detection, have been designed to address such goals and to prevent attacks related to privacy, non-repudiation and man-in-the-middle. The third type of security vulnerabilities is specific to virtual networks themselves. These could arise when these networks are programmable. In the absence of well-structured policies and rules, programmability could significantly increase the vulnerability of the network. So far, security issues that are specific to virtualized networks are relatively unaddressed in the field in network virtualization. We have neither proved that virtualized networks are as secure as traditional networks, nor provided enough security measures to defend these new innovations. This field certainly requires careful scrutinization.

In this chapter, we present our understanding of network virtualization. The extension of the SILO architecture for network virtualization is presented in the next chapter.

Chapter 7

SILO Extension for Network Virtualization

From the last chapter's discussion, we define network virtualization as any form of partitioning or combining a set of network resources, and presenting it to users such that each user can use the partitioned or combined resources and have a unique separate view of the network. Resources can be fundamental (nodes, links and addresses) or derived (topologies).

In this chapter, we present how the SILO architecture embraces network virtualization. In Section 7.1, we present our basic reasoning, "Virtualization as Service." In Section 7.2, we detail the SILO architectural extension for network virtualization and new concepts derived from this extension.

7.1 Virtualization as Service

So far, network virtualization has been strongly coupled with the platform and hardware of the substrate. Logically, however, network virtualization consists of many coordinated individual virtualization capabilities, distributed over networking elements that share the common functionality of maintaining resource partitions and enforcing them. In keeping with the SILO vision, we can view these functions as separate and composable services.

The most basic of these services are those of *splitting* and *merging* flows; these services must obviously be paired. This is no more than the ability to mux/demux multiple contexts. Note that this service is a highly reusable one and can be expected to be useful in diverse scenarios whenever there is aggregation/dis-aggregation of flows, such as mapping to/from physical interfaces, at intermediate nodes for equivalence classes on a priority or other basis, or label stacking.

In the networking context, virtualization is usually interpreted as implying two capabilities

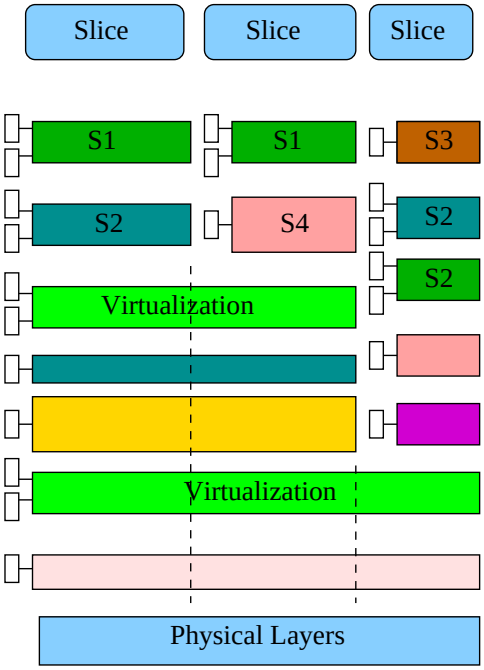

Figure 7.1: Successive Virtualization

beyond simple sharing. The first is *isolation:* each user should be unaware and unaffected by the presence of other users and should feel that it operates on a dedicated physical network. This is sometimes also called "slicing." This can be broken down into two services: (i) slice maintenance, which keeps track of the various slices and the resources used by them, and (ii) access control, which monitors and regulates the resource usage of each slice, and decides whether any new slices requested can be created or not; for example, rate control such as leaky bucket would be an access control function.

The second capability is *diversity:* each user should be able to utilize the substrate in any manner in which it can be used, rather than being restricted to employing a single type of service (even if strictly timeshared). This is akin to the ability to run different operating systems on different virtual machines. In SILO, this capability is natively supported through the composable nature of the stack. Not only do different silos naturally contain different sets of services, but the composability constraints provide a way to indicate which set of upper services may be chosen by different slices when building a particular virtualized substrate.

The definition of a standard set of services for virtualization means that every realization of this service (for different substrates) would implement the functional interfaces specified by the service itself; thus, any user of the virtualization agent would always be able to depend on these standard interfaces. Articulating this basic interface is part of our goal in this regard. For

example, consider the case of virtualizing an 802.11 access point with the use of multiple SSIDs; the interface must allow for the specification of shares, similar to the example above. However, as the different slices can use varieties of 802.11 with different speeds, the sharing must in fact be specified in terms of time shares of the wireless medium, which is the appropriate sharing basis in this context.

7.1.1 Generalizing Virtualization

Following the principle that a virtual slice of a network should be perceived just like the network itself by the user, we are led to the scenario that a slice of a network may be further virtualized. A provider who obtains a virtual slice and then supports different isolated customers may desire this scenario. The current virtualization approaches do not generalize gracefully to this possibility because they depend on customized interfaces to a unique underlying hardware. If virtualization is expressed as a set of services, however, it should be possible to design the services so that such generalization is possible simply by re-using the services (see Figure 7.1). There would obviously be challenges and issues. One obvious question is whether the multiple levels of virtualization should be mediated by a single SMA or whether the SMA itself should run within a virtualization, and thus multiple copies of SMA should run on the multiple stacks. Either approach is possible, but we believe the former is the correct choice. In OS virtualization, the virtualization agent is itself a program and requires some level of abstraction to run, though it maps virtual machine requests to the physical machine to a high level of detail. Successive levels of virtualization with agents at all levels being supported by the same lower level kernel are difficult to conceive. However, networking virtualization agents do not seek to virtualize the OS that supports them. As such, the kernel support they require can be obtained through a unique SMA.

It may appear from this discussion that with per flow silo states, there is no need to virtualize, and in fact, it is possible to extend all the slices to the very bottom (dotted lines in Figure 7.1). However, the advantage lies precisely in state maintenance; a service that is not called upon to distinguish between multiple higher level users can afford to maintain its state only for a single silo, and the virtualization service encapsulates the state maintenance for the various users.

7.1.2 Cross-virtualization Optimization

Finally, it is possible to conceive of cross-layer interaction across virtualization boundaries, both in terms of composability constraints and tuning. The service S1 in Figure 7.1 may require the illusion of a constant bit-rate channel below it, and the virtualization below it may provide this by isolation. If, however, there is some service still further down that does not obey this

restriction (some form of statistical multiplexing, for example), then the illusion will fail. It must be possible to express this dependence of S1 as a constraint, which must relate services across a virtualization boundary. It appears harder to motivate the need to tune performance across boundaries, or even (as the SMA could potentially allow) across different slices. Although we have developed some use cases, they are not persuasive. However, we recall that the same was true of the layering abstraction itself, and it is only recently that cross-layer interactions have come to be perceived as essential. We contend that cross-virtualization optimization is also an issue worth further investigation, even if the motivation is not apparent now.

7.2 Architecture Extension for Network Virtualization

7.2.1 Extension of Endpoint

When we attempted to introduce sharing, isolation and diversity into the prototype framework, we unexpectedly discovered that the very first extension needed is the endpoint.

Conceptually, an endpoint remains where a silos end or starts. However, an endpoint might not always be the boundary between a silo and the outside world. It could be the boundary between silos. It must support merging or splitting silos.

As shown in Figure 7.2, we introduce the concept of ports in an endpoint. Silos are connected to those ports.

As mentioned previously, services are where the functionalities come from in the SILO architecture. We do not want to design some "intelligent" endpoint, which can decide the merge or split of silos. We retain these functions in services, such as Service F and Service L in Figure 7.2. The endpoint is merely a connector that provides ports for silos.

7.2.2 Siloplex

With the extension of the endpoint, we expand from the original single, top-to-bottom silo or from a set of single silos, to a siloplex. Figure 7.2 shows a typical siloplex. The upper silo splits into two silos at the bottom. The endpoint with three ports, P1, P2 and P3, acts as the connector of these three silos.

7.2.3 Virtualization Service

We envision various virtualization services discussed in Section 7.1. In silos, virtualization services are placed to where resources are shared.

In Figure 7.2, Services F and L are these virtualization services. Service L splits and merges the dataflow; Service F maintains isolation from other services and aids Service L's splitting and merging.

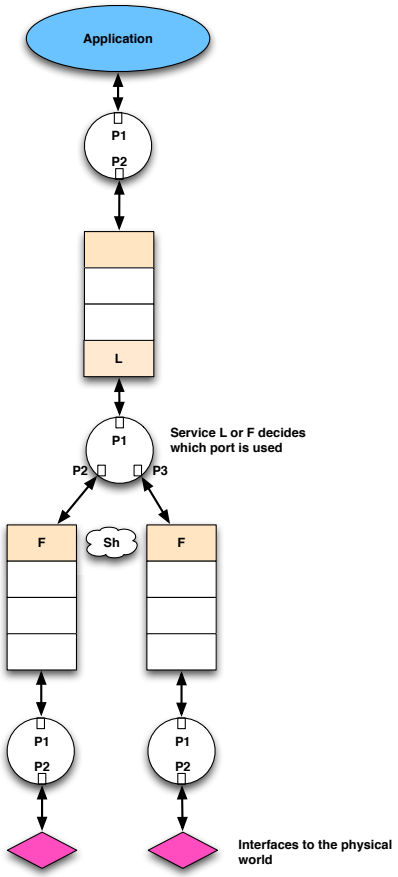

Figure 7.2: Siloplex

We have implemented the Virtual Concatenation (VCAT) service with the Link Capacity Adjustment Scheme (LCAS) tuning algorithm in this extended architecture. Please refer to [102] for additional details.

In this chapter, we explore the architectural extension for network virtualization. In the following chapter, we present the integration of the SILO architecture with other architectures in GENI.

Chapter 8

IMF: Integrating SILO with Other Architecture in GENI

In previous chapters, we described the SILO framework and detailed its components. In this chapter, we continue to explore another important architectural aspect: integration with other existing architectures. We describe "Integrated Measurement Framework," a.k.a IMF. IMF is one of the GENI projects; SILO itself is an important component in IMF and interacts with the control framework and the measurement plane. The IMF project is a joint effort by North Carolina State University, Columbia University and RENCI.

8.1 Background of IMF

In this section, we introduce the necessary background in order to present our IMF project clearly. The background includes the Global Environment for Network Innovations (GENI), the Open Resource Control Architecture (ORCA), and the Breakable Experimental Network (BEN).

8.1.1 GENI

As we discussed in Section 6.5.2, Global Environment for Network Innovations (GENI) is a U.S. initiative sponsored by the National Science Foundation (NSF), with the aim of creating network facilities in order to host advanced network research.

Some terminologies, such as Guest, Host, Sliver, and Slice, are widely used in our project and in other GENI-related projects. Figure 8.1 shows a guest-host model. Guest is the user of the resource. Host is the provider of the resource and is able to provide resources by the unit of Sliver. Slivers from different hosts can be combined as a slice and eventually presented to the guest.

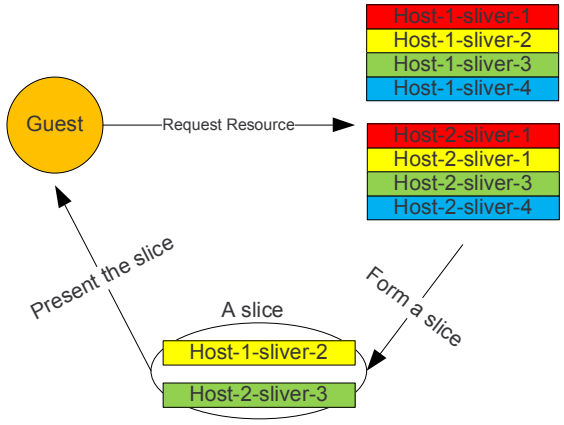

Figure 8.1: Guest-host Model: Guest, Host, Sliver, and Slice

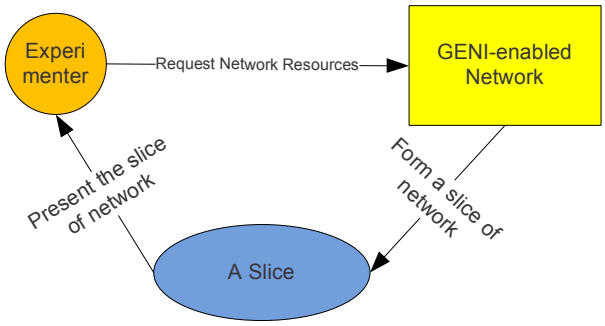

Figure 8.2: GENI Guest-host Model

The purpose of GENI can be expressed by this guest-host model perfectly. As Figure 8.2 shows, in the world of GENI, guest is an experimenter who would like to perform a test on a slice of network provided by a GENI-enabled network.

GENI sponsors a collection of research projects, most of which are clustered around four control frameworks, PlanetLab, ProtoGENI (descendent of Emulab), ORCA (Open Resource Control Architecture) [99, 100] and OMF (Orbit Management Frmaework) [101].

We have discussed PlanetLab and Emulab, when we enumerated some popular network testbeds in Section 6.3.1. There are two main components of these testbeds: the control framework and network/computing devices. When PlanetLab and Emulab were initially designed, they were aimed to provide deliverable testbeds. As a result, their control frameworks are highly coupled with particular devices, particular configurations, and a particular implementation. On the other hand, ORCA was designed as a control framework itself and does not make assumptions regarding what types of resources or what types of testbeds it will control.

Under the GENI umbrella, most projects are clustered around four frameworks mentioned

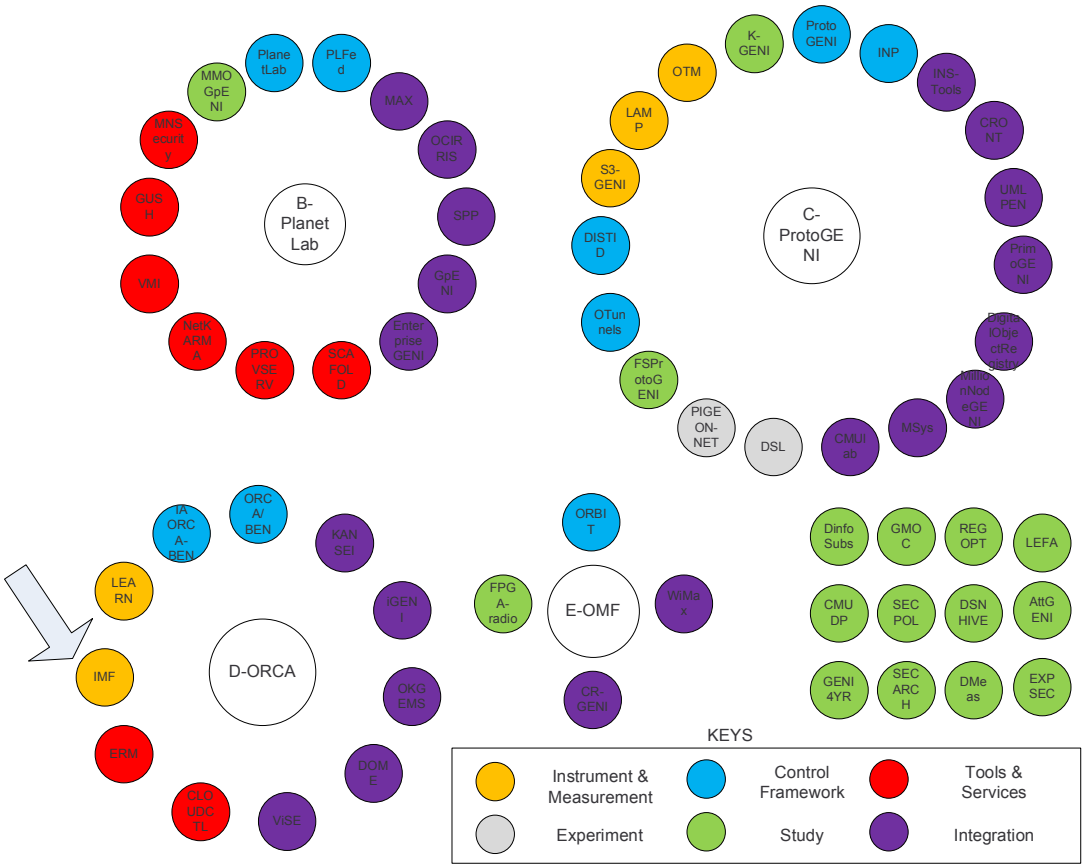

Figure 8.3: IMF in GENI (adapted from the GEC new comer slides)

previously. Most projects focus on a particular aspect, including control framework, instrumentation and measurement, experiment workflow and service, etc.

Our "SILO in GENI" project, a.k.a. IMF, is clustered around the ORCA framework and mainly focuses on network measurement. Figure 8.3 shows IMF's place with GENI. This figure is updated by the GENI Project Office regularly and can be found in the newcomers' session at GENI Engineering Conferences (GEC) [103], which are held every four months.

For more details about GENI, please visit GENI's official Web site [13]. GENI Engineering Conferences are also open to the public.

8.1.2 ORCA

Open Resource Control Architecture (ORCA) is a resource management framework. It is developed by Duke University and currently maintained by RENCI and Duke University. Unlike its counterparts in PlanetLab and ProtoGENI (Emulab), ORCA is a generic resource management

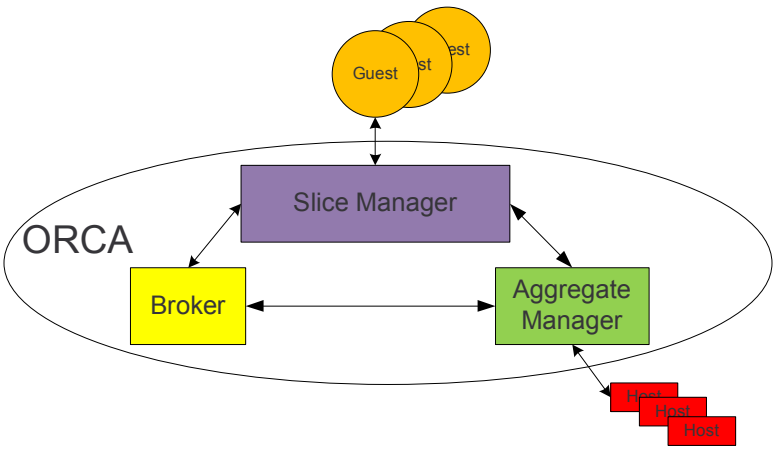

Figure 8.4: ORCA

framework without any particular assumptions regarding types of resources.

ORCA enables the guest-host model shown in Figure 8.1 and abstracts the resource leasing concept. In ORCA, a request for resources is abstracted as a ticket. If the resources are available, the ticket can be redeemed. There are three actors in ORCA: Slice Manager, Broker, and Aggregate Manager.

Slice Manager - This actor provides interfaces to guests, and creates, configures, and adapts slices.

Broker - Broker coordinates and schedules resources.

Aggregate Manager - Aggregate Manager is also called Authority and provides access to resources.

Figure 8.4 shows three actors of ORCA and ORCA's position in the guest-host model. For a detailed description of ORCA, please refer to "the ORCA book" [104], ORCA paper [100], and the official ORCA Web site [105, 99].

8.1.3 BEN

Breakable Experimental Network (BEN) is a metro-scale optical testbed connecting the campuses of RENCI, North Carolina State University, University of North Carolina at Chapel Hill, and Duke University. It is primarily maintained by RENCI.

BEN is supported by three universities, RENCI and MCNC MCNC, all of which aim to provide researchers with an exclusive network to perform experiments. This philosophy perfectly echoed GENI's guest-host model. BEN provides hosts of resources, and it indeed requires

a control framework to fill the vacancy of the control part. ORCA, as a generic control framework, is the perfect candidate for BEN. Through deployment in BEN, ORCA in turn has been pushed to a more mature stage.

ORCA-BEN represents ORCA-controlled BEN, which is also a GENI project. ORCA-BEN aims to provide slices of BEN from the physical layer to the application layer. For the rest of this thesis, when we mention BEN, we indicate ORCA-BEN unless we explicitly say otherwise.

From the hardware perspective, we deployed Polatis fiber switches, Infinera DTNs, Cisco and Juniper IP routers and several data centers in BEN. From the software perspective, we deployed a DNS server, XMPP server, OpenVPN server, KVM-enabled and Xen-enabled Eucalyptus Private Cloud [106], etc.

Using Virtual Machine technology is an easy way to slice a physical machine, and a computer cloud is an easy way to slice a group of machines. Unlike PlanetLab and ProtoGENI, we adapt the de facto, industrial standard, Amazon-cloud-compatible Eucalyptus Cloud instead of developing proprietary software to manage the computing clouds.

8.2 Integrated Measurement Framework

Integrated Measurement Framework (IMF) is a project that investigates real-time network performance measurements and uses the measurements to dynamically configure the network for better performance. As Figure 8.3 shows, IMF chose ORCA-BEN and clustered around ORCA.

In terms of the guest-host model, the SILO architecture in IMF resembles the guest; BEN is the host; and ORCA is the control framework.

8.2.1 IMF Overview

GENI-IMF (GENI Integrated Measurement Framework) is an extendible component-based architecture aimed to serve the experiment data collection needs of GENI users. It provides an abstraction of measurement capabilities available within the GENI substrate by allowing experimenters to measure properties within the substrate, such as physical layer attributes (e.g., optical or RF power) and performance parameters (e.g., BER, packet loss, CPU utilization) through a unified measurement interface. IMF interoperates with the GENI CF (Control Framework) in order to (a) instantiate appropriate measurement functions into the slice and (b) provide authorized access to these measurement functions to user tools (see Figure 8.5). IMF also interoperates with GENI-fied storage services in order to store experiment data long term.

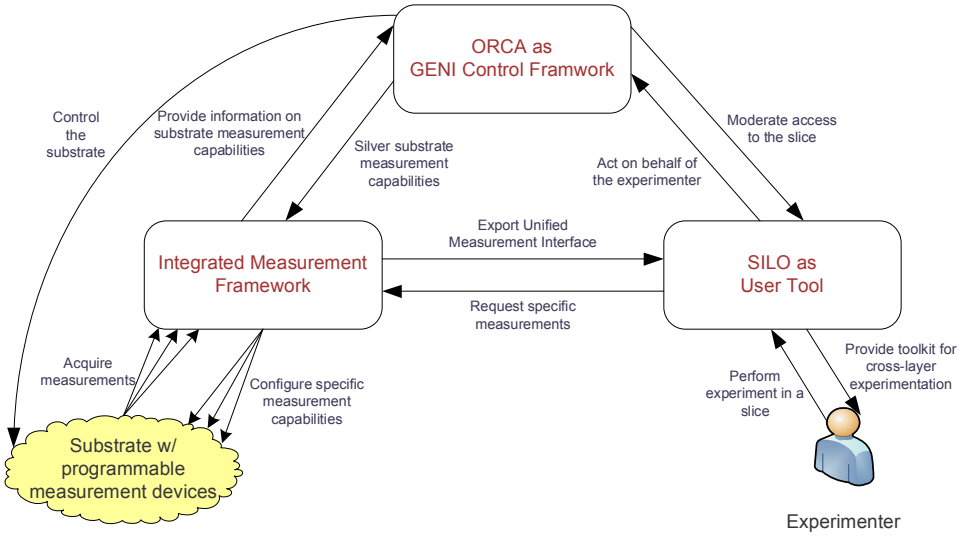

Figure 8.5: IMF Overview

8.2.2 IMF Architecture Overview

IMF operates in an environment consisting of multiple sites, where each site belongs to a single administrative domain and contributes some resources to the GENI infrastructure. Each site possesses a physical or virtual management network. The sites' management networks are linked together by the commodity Internet, which also provides connectivity between the elements of the control framework, IMF and users. The users of the system are GENI experimenters. At the user's request, the control framework creates an experiment slice by instantiating a set of slivers of resources from multiple sites and interconnecting them. The created slice has a data plane, which reflects the topology desired by the user for the experiment. The measurement plane of the slice represents the measurement functions configured by the control framework and available within the slice and elements of IMF configured to collect, store and distribute the measurement data from these functions. These capabilities are made available to the user through the Experimenter Tool. A user can also integrate these capabilities into the experimental stack using the SILO interface, which provides the ability to control and manipulate the measurement capabilities from the experiment environment (see Figure 8.6).

There are several types of consumers of measurement data:

- User/experimenter tools existing outside the slice (ET);
- In-slice functions that operate on the measurement data and react as per their functionality, possibly in order to provide a closed feed-back loop (SILO); and
- Storage functions that collect and store the measurement data for later retrieval; IMF

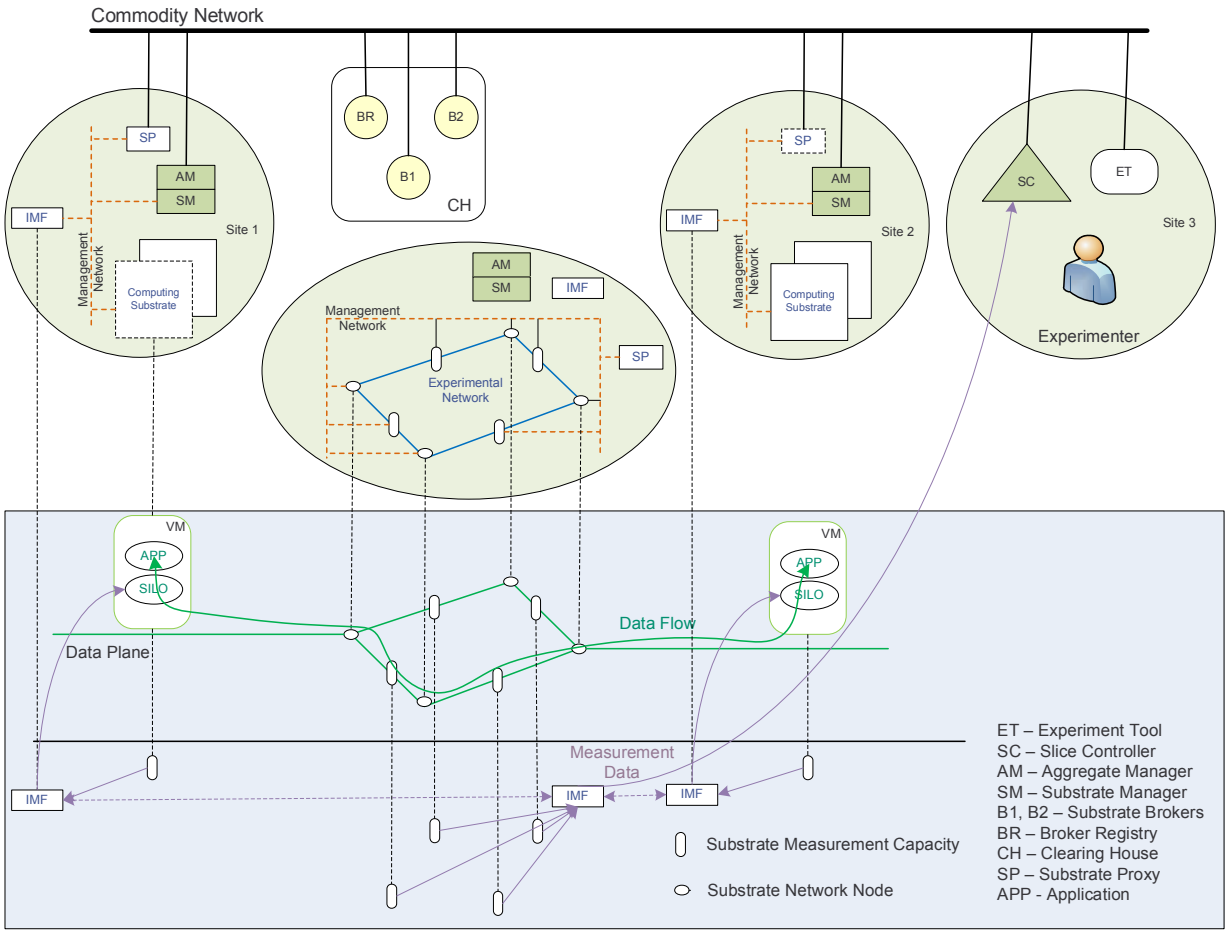

Figure 8.6: IMF Architecture

provides authenticated access and enforces policy-based access to the measurement data.

A consumer of measurement data can be interested in all measurement data from a given slice, measurement data only from a specific type of measurement function, or data from only a specific site, whose resources are part of the slice. Finally, a consumer can be interested in measurement data from only a single instance of a measurement function out of the many instances available within a slice.

IMF is distributed across multiple sites (a site can have none, one or more instances of IMF). IMF instances federate in order to make measurement data from slices available to the consumers. Consumers can connect to any instance of IMF to receive the desired measurement data. IMF is implemented as a set of federated servers implementing publish/subscribe or pubsub functionality. Considering the potentially large number of users of measurement data, who can be interested in diverse subsets of measurement data generated in a slice, pubsub offers

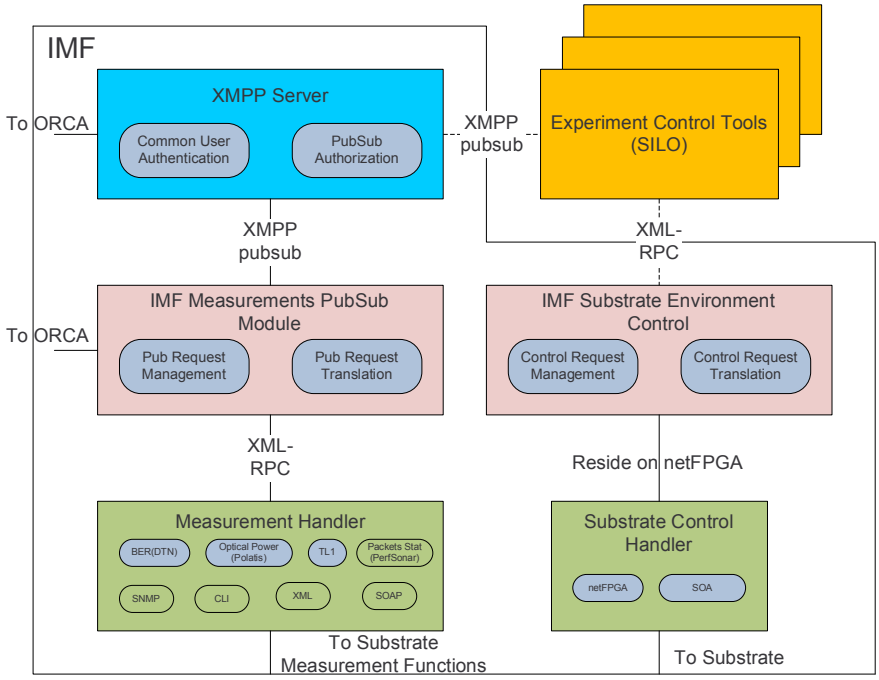

Figure 8.7: IMF Components

scalability advantages over other approaches. Using pubsub, a measurement data consumer registers interest (subscribes) only to the measurement data it is interested in and receives this data at the frequency that the data is generated, thus removing the need for explicit polling by data consumers.

Substrate Environment Control

An important function of GENI is establishing a proper environment for the experiment in the substrate. The example of the environment may be a certain level of attenuation in optical links, or a certain level of interference in RF experiments. In IMF, this function is tightly coupled with measurements to guarantee experiment repeatability and that only users authorized to manipulate the slice can do so.

8.2.3 Component Architecture

The foundation of the IMF is a mature set of XMPP standards [107, 108, 109] and existing software implementations [110, 111, 112], defining and implementing pubsub functionality.

The component architecture in Figure 8.7 is organized around the *XMPP server* that (a) tracks subscription requests from measurement data consumers and (b) receives publish events

from the *IMF Measurement Pubsub Module*, which contains measurement data or meta-data. The *IMF Measurement Pubsub Module* interacts with the substrate by polled or interrupt-driver I/O using the *Measurement Handler*. The *Measurement Handler* contains the low level mechanisms and logic necessary to interface with substrate measurement functions, and it plays a role equivalent to device drivers in an OS in the IMF. At this time, the intent is for publications to originate only from the Measurement handler, and subscriptions only from Experimenter Tools or SILO (though the mechanism is general, allowing future extensions if needed).

XMPP Server

The main functions of the XMPP server are to:

- Maintain and allow authorized updates of the published information structure (as per [109] structured as "nodes");
- Interoperate with common GENI authentication mechanisms to authenticate users of the measurement data; and
- Maintain and consult with authorization policy storage to provide policy-based access control to the published measurement data.

IMF Measurement PubSub Module (PSM)

The main functions of the pubsub module are to:

- Translate information between the physical topology of the substrate and the virtual topology of the slice, as perceived by the user and experiment control tools. PubSub requests from the control framework are translated from the virtual topology to the physical substrate. Published data are translated from the physical topology into the virtual topology of the slice;
- Manage publish requests created by the CF and interface with the measurement handler; and
- Manage subscriptions, such as join and leave events.

IMF Substrate Environment Control Module (SEC)

The main functions of the Substrate Environment Control Module are similar to the PSM module, when applied to substrate environment control as opposed to measurement data collection.

Measurement Handler (MH)

The Measurement handler presents an XML-RPC interface to the pubsub module in order to be separable from it. The MH:

- Presents a uniform interface to configure and query substrate measurement capabilities; and
- Maintains the necessary low-level libraries and logic to interface with a diverse set of hardware and software representing the substrate measurement functions.

Substrate Control Handler (SCH)

An additional module shown in Figure 8.7 is the Substrate Control Handler, which presents a uniform interface for functions and components within the substrate that can help manipulate the experiment environment. This will provide the experimenter with an increased ability to adjust the network resources and/or devices to test the functionality of his/her experiment. For example, in an RF environment, a set of signals can be introduced in order to test a particular wireless protocol for susceptibility to RF interference. In an optical environment, a programmable attenuator may reduce the optical signal power on a given link to help test a new FEC implementation. These experiment environments may be manipulated at the start of an experiment for initial setup, or they may be dynamically altered during an experiment based on real-time data (such as measurement information obtained from the measurement handler). Experimenter tools can drive these functions via the Substrate Control handler.

Measurement Data Consumer Communication Patterns

IMF supports different communication patterns, depending on the capabilities of the substrate measurement functions, users and their tools, and the properties of the measurement data. These patterns are shown in Figure 8.8.

Pattern (1) is the simplest, in which IMF provides a complete separation between the measurement functions and the consumers. The MH serves as the immediate interface between the substrate measurement function and the rest of the world. The MH configures itself for data collections and serves as a consumer of the data, which it then publishes to the XMPP server. This case is common for non-GENI-fied measurement functions that provide moderate amounts of record-structured measurement data at intervals.

In pattern (2), the MH can instruct the measurement function to publish data to the XMPP server, but does not play a role in the publishing process. This pattern is suitable for intelligent measurement functions supporting the XMPP pubsub protocol.

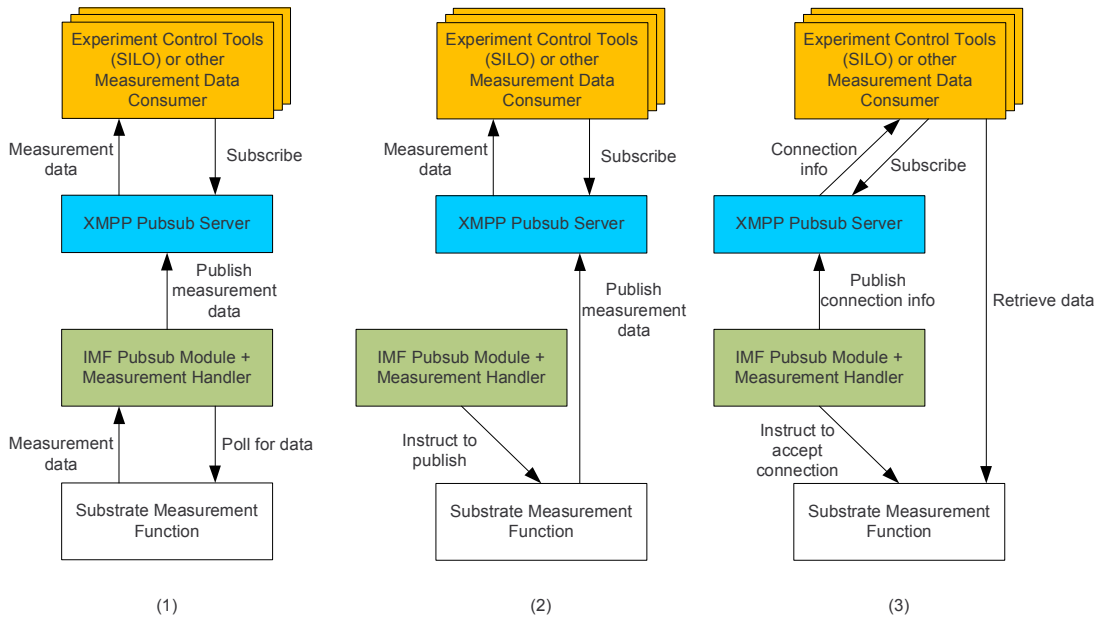

Figure 8.8: IMF Communication Patterns

Finally, in pattern (3), the MH instructs the measurement function to listen for incoming connections from user tools with a specific protocol (the simplest could be 'Open a TCP server socket on port X') and publishes the meta-information about the data (i.e., that it can be found on port X on a specific server) to the XMPP server. User tools can then directly access the measurement data from the measurement function. This pattern is most suitable for supporting measurement functions that generate large volumes of data (i.e., packet capture).

8.2.4 Enabling Protocol Use of Measurements with SILO

As mentioned before, consumers of the measurement information include those outside the slice, as well as a programmatic entity inside a slice. An example of the former type is a measurement console application for the user, which enables users to decide conditions or parameters for the experiment. The second type of consumption is needed when a user wishes to design an experiment in which one of more protocols in the stack are experimental ones intended to react to measurement data dynamically during operation, for example, a connection routing algorithm that selects routes for incoming connection requests based on recent measurements of signal strength or optical fiber signal quality (such as PMD).

Figure 8.6 shows the position of the SILO services in the data plane. As indicated in Figure 8.7, the interaction with the other IMF components is the same for SILO services as when a passive Experimenter Tool (such as a measurement console application) is the consumer.

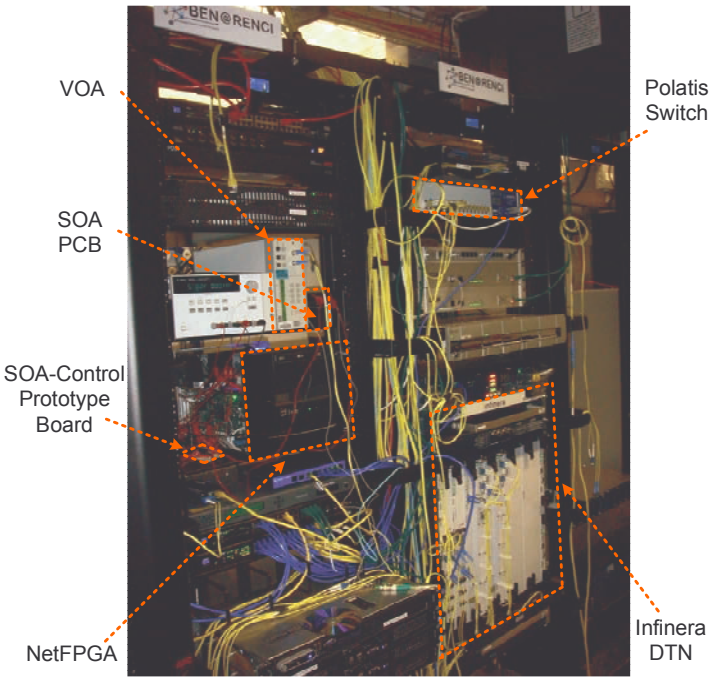

Figure 8.9: IMF Physical Infrastructure

8.2.5 Interactions with the Control Framework

The control framework (ORCA) creates slice and measurement capabilities within it, based on requests from the experimenter. After creation, the Control Framework instructs PSM to interface to the substrate via MH to acquire measurement data. Then, PSM publishes data to the XMPP server.

8.3 Cross-Layer Experimentation with SILO

8.3.1 Experimentation Setup

As Figure 8.9 shown, the experimentation is set up in BEN at the RENCI site. The Polatis optical switch and the Infinera DTN are part of BEN. Variable Optical Attenuator (VOA) is used to introduce power lose on the link. It can be manually set or automatically adjusted by a script. The Semiconductor Optical Amplifier (SOA) PCB, the SOA-control prototype board, and the NetFPGA compose of a remotely controlled, tunable SOA. It can be used to compensate power loss on the link.

Figure 8.10 shows the physical connections between optical devices in BEN at the RENCI site. The upper half shows the logical connections, and the lower half shows the physical

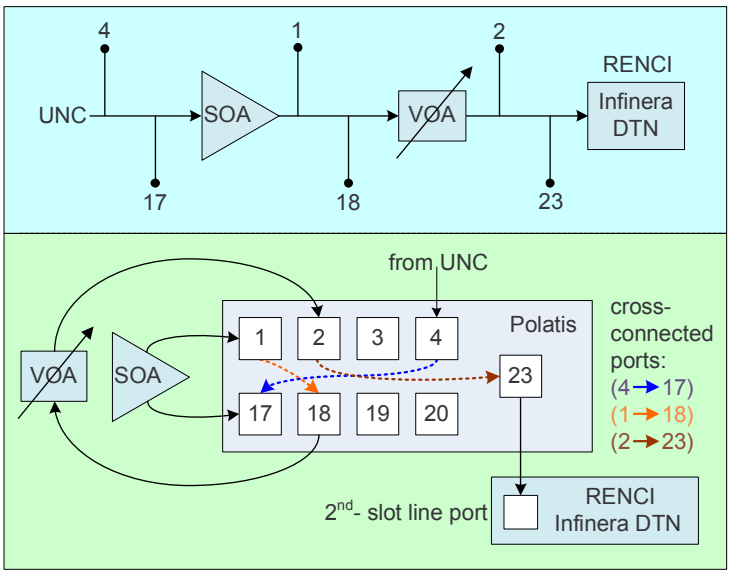

Figure 8.10: IMF Physical Configuration

connection and cross-connected in the Polatis Optical switch.

Figure 8.11 shows the VLAN configuration in BEN at the RENCI site and the UNC site. SILO runs on the virtual machines in the Eucalyptus clouds. Virtual machines in the both sites are manually configured into a VLAN. VLAN translation technology is used to achieve this on the Cisco 6509 in the both sites.

8.3.2 Cross-Service Control and Video Stream Demo

Figure 8.12 shows the details of the cross-service control and video stream demo. At least four SILO services are used in the each side.

Packet Counter - Counts the number of packets;

Measurement Collector - Collects port power/BER from Polatis and Infinera, published by the XMPP server;

SOAC - Controls the SOA through XML-RPC; and

Interface Switch Switches video stream to the reference path when the SOA/VOA controlled path cannot be compensate anymore.

A SILO algorithm is loaded by the Silo Tuning Agent. Video monitor and Interface Switch algorithm oversees all four services, and tune the knobs, and try to reach three goals:

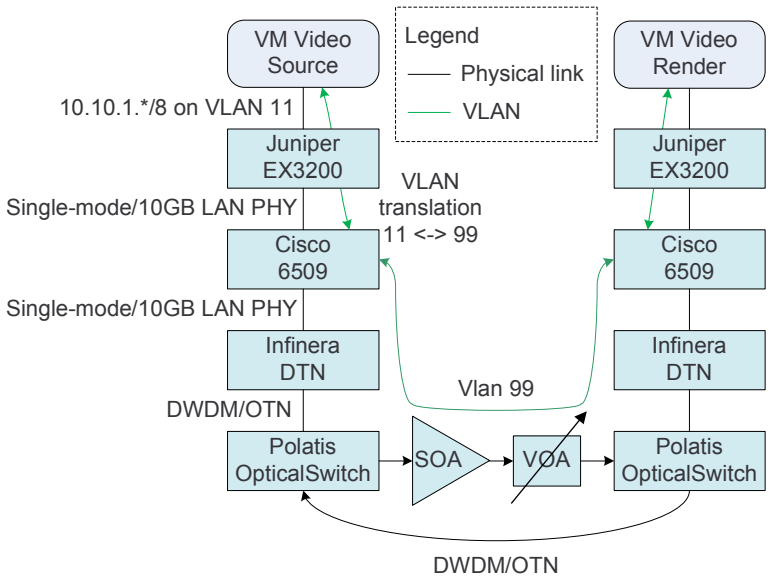

Figure 8.11: IMF Virtual Machine VLAN Configuration

(1) When the power decreases because of the attenuation (controlled by the Matlab script), increase the SOA's gain to compensate the power loss;

(2) Make sure the port power stay in the safe level, and decrease the SOA's gain when necessary;

(3) When the SOA/VOA controlled path cannot be compensated anymore, make decision to switch all traffic to the reference (backup) path. As currently there is no available reference path in BEN, SILO switch traffic to the management network.

8.3.3 Interfaces between SILO and NetFPGA

The NetFGPA card controls the SOA through the physical pins shown in Figure 8.12. It is a PCI card on a PC running CentOS. A XML-PRC server runs on this Linux PC. The interfaces between the SILO SOA Controller service and the XML-RPC server have been defined and implemented.

Originally, we tried to design functions like set_gain(x dbm), get_gain(x dbm). However, we found out it was not feasible because neither the SOA nor the netFPGA does maintain any gain status. Considering this, we use the SILO SOA Control service to maintain the gain status, and the following functions are designed.

soa_up : increases the SOA's gain by the minimum granularity. This minimum granularity is not liner, and it changes between 0.3 dBm and 0.6 dBm.

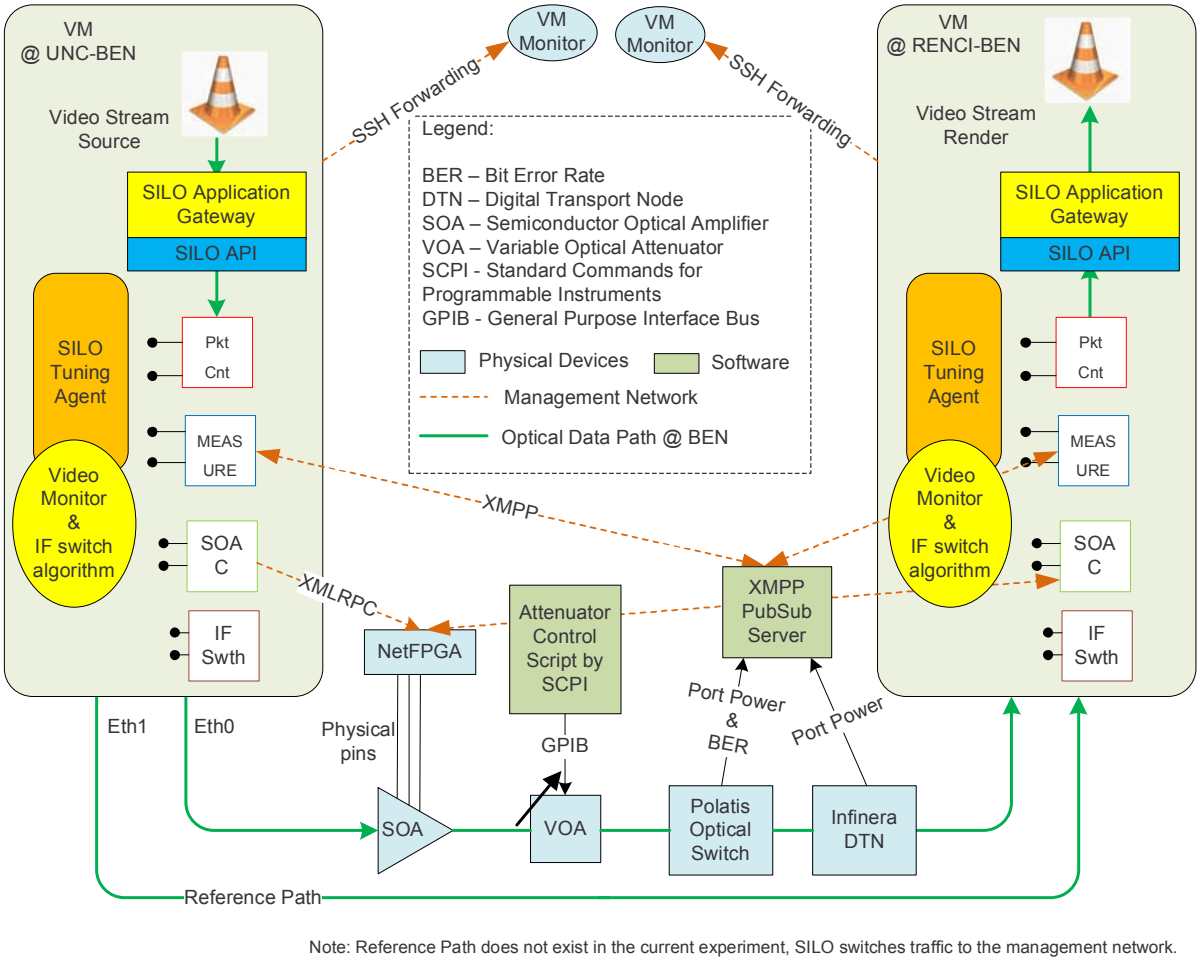

Figure 8.12: IMF Cross-Service Demo

soa_down : decreases the SOA's gain by the minimum granularity. This minimum granularity is not liner either, and it changes between 0.3 dBm and 0.6 dBm.

get_soa_gain_threshold : (optional) gets suggested value of the maximum gain that the SOA is able to achieve. This value is preset in the XML-RPC server. The SILO service should be able to use this value in the tuning algorithm to make sure the SOA is not overloaded.

The SILO SOA Controller service uses the xmlrpc-c library [113].

8.3.4 Results

Figure 8.13 shows the baseline of our experiments. The attenuator fluctuates from 0 dBm to 10 dBm in the manner of a step function. The output of the attenuator changes 1 dBm every 10 seconds.

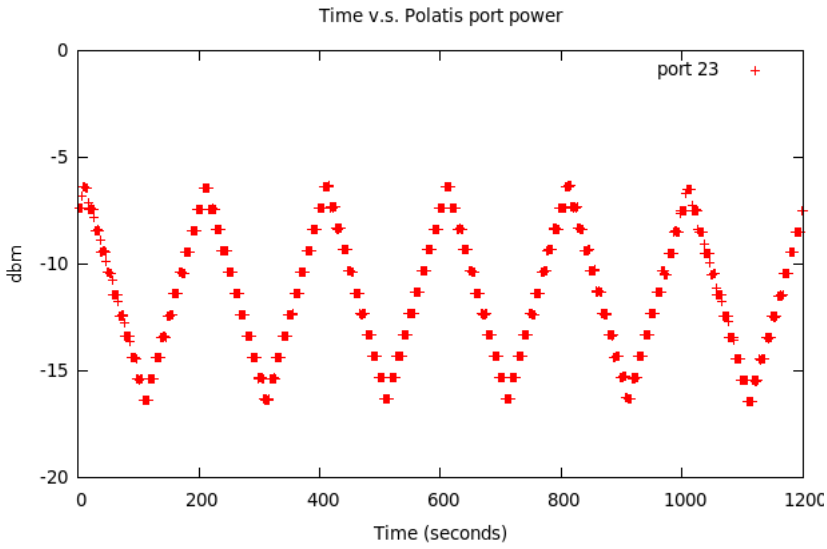

Figure 8.13: Port Power w/ Attenuating Fluctuation of 10 dBm

Figure 8.14 shows the result when we introduce tuning algorithm to maintain the port power around -11 dBm. Whenever the port power (monitored by the measurement service) is beyond this scope, the SOA Controller service increases or decreases the SOA's gain by calling soa_up or soa_down one time. It shows port power remains between -10 and -13 dBm most of the time.

The algorithm used in Figure 8.15 decreases or increases the SOA's gain by calling soa_up or soa_down x times (x=(current port power - 11)/0.6). This algorithm assumes that one call of soa_up or soa_down changes the gain by 0.6 dBm. As the fluctuation of the attenuator is always 1 dBm every time, the result is roughly the same as the previous result shown in Figure 8.14.

The tuning algorithm could be further improved considering the SOA's power vs. gain relation.

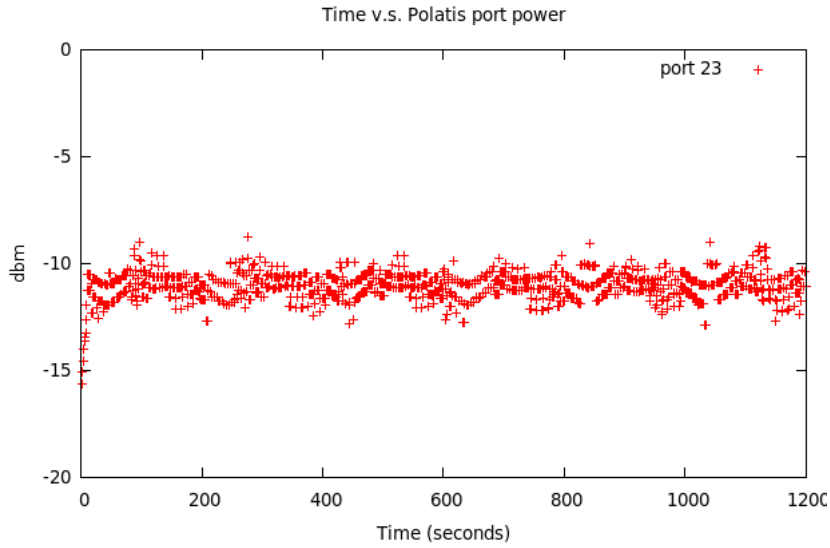

Figure 8.14: Port Power w/ Attenuating Fluctuation of 10 dBm w/ SILO Tuning (1)

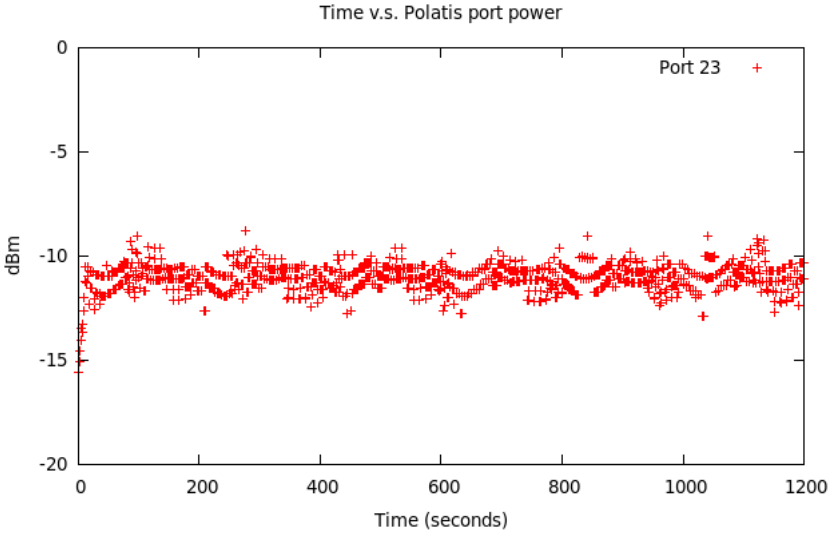

Figure 8.15: Port Power w/ Attenuating Fluctuation of 10 dBm w/ SILO Tuning (2)

Chapter 9

Summary and Future Work

9.1 Summary

In this work, we proposed the SILO architecture in our effort to explore the nature of the future Internet. Following our philosophy "designing for change," we presented our design principles, the fundamental design and components of the SILO architecture. By comparing this with other similar composable architectures, we reveal the benefits of the SILO architecture.

We then present two main benefits of the SILO architecture: (1) cross-service tuning and (2) architectural support for customized SILO services, tuning algorithms, and applications. We detail how we enable these features from the architectural point of view.

We also extend the SILO architecture for two important aspects: (1) network virtualization and (2) integration with other architectures in GENI. These extensions keep the SILO architecture competitive and fulfill our philosophy of "designing for change."

From our experiments and positive feedback from users, we are confident the SILO architecture is promising and will be the cornerstone for our future research.

In the following of this chapter, we present our comprehensive vision of the future Internet and highlight the contribution of the SILO architecture to this vision in Section 9.2. Then, we discuss the future directions to explore in Section 9.3.

9.2 Our Vision of the Future Internet: ONE

The Internet has enjoyed success since its inception in the 1950s and has changed almost every aspect of society. With the open standards of most commodity Internet protocols and the promise of Internet neutrality, we have entered a new age of technology.

However, compared with its open standards and neutrality, the implementations of the Internet remain relatively closed. Routers and switches carrying the network traffic currently

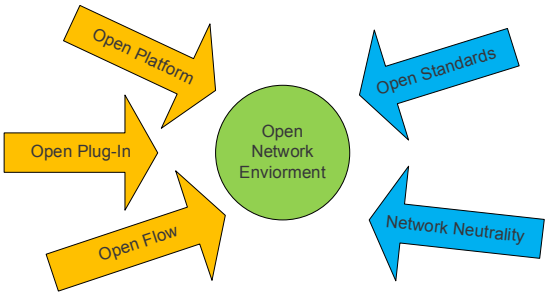

Figure 9.1: Open Network Environment

run on manufacture-customized software and hardware; it is difficult for users or third parties to plug in their own software or hardware modules into the Internet; and the data flow is predetermined as long as it enters the networks.

Besides the open standards and network neutrality, we envision more openness (open platform, open plug-in, open flow) will be introduced to the Internet, which would then become the Open Network Environment (ONE). Figure 9.1 shows five aspects that we recommend for the Open Network Environment.

Open Platform Most traditional commodity routers run customized operating systems, such as Cisco's Internetwork Operating System (ISO), VxWorks, on manufacture-customized hardware. There has been a trend that Linux has been adopted as one of the major players of OS for network devices.

A goal of Open Platform is providing a unified operating system that can run upon generalized hardware, such as standard X86 architecture. A list of router software distributions is provided in [49]. Vyatta [50] is one example where generalized router software is provided, and it directly competes with Cisco and other router manufactures in the low-end router market. HP Lab has developed an Open Router Platform [114] and attempts to generate common interfaces for various hardware abstractions.

Open Plug-in Even though the Internet is designed openly with open standards, it is not easy at all to plug in a particular protocol implementation or new functionalities. When the opportunities of inventions are controlled by several large players, the inventions themselves are jeopardized. Several recently successful software continues to remind us that innovation is unlimited once the door is opened to the public. Thousands of Apple iPhone applications, Facebook plug-in Webs, and Firefox plug-ins enrich the corresponding products day by day. The Unified Network Platform [115] is an attempt to modularize network components and to realize almost a plug-in ability.

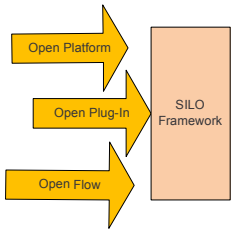

Figure 9.2: SILO: Convergence of Paths

Open Flow Early philosophy [1, 2] directing the design of the Internet resulted in a dumb net with smart ends. Once the data enter the Internet, the data follow predefined routes to the destination without modification. We simply lose control of the data flow. The Openflow project [116] is an attempt to address this problem and to gain control of the data flow while still not disrupting the basic design philosophy.

The SILO architecture has been our effort to provide a convergent way to the Open Network Environment, as shown in Figure 9.2. As our "design for change" physiology indicates, the SILO architecture is a relatively open architecture.

(1) It is built on the open platform, open source, and no particular hardware dependency.

(2) It gracefully supports SILO services and tuning algorithms' dynamic plug-ins.

(3) It enables applications to create flexible silos, resulting in flexible data flow.

9.3 Directions to Explore

The SILO architecture is on its way to maturity, and there are several directions that future research can take.

(1) **SILO in the core network**: Our prototype framework is currently supported in the hosts, although we always envision that the SILO architecture should be deployed across the network. There are a number of unexplored areas for the SILO architecture in the core network.

(2) **Negotiation between the SILO architectures**: The current negotiation process between hosts running the SILO prototype is relatively simple. The SILO architecture does not define how these negotiations work either. This work is critical when we have large amounts of the SILO architecture deployed in a large scale network. Although some might

argue that the negotiation should be completed by services, we believe services are mainly for the data plane, and the negotiation in the control plane is still unreplaceable.

(3) **Interconnection with the current Internet**: We originally designed the SILO architecture with the clean-slate in mind, but gradually, we moved to the middle ground between revolution (clean-slate) and evolution (compatibility). Exploring how the SILO architecture could be compatible with the current commodity network will help further deployment, and we think this is a worthy direction of research to explore.

There is also another direction not directly related to the SILO architecture, but we consider it is a vast area worthy of considerable explorations: **Network Resource Presentation for Virtualization**. Although the Resource Description Framework (RDF) and the Web Ontology Language (OWL) might be used, we believe, in the wake of the recent resource presentation workshops in the GENI Engineering Conferences, they are inadequate to present network resources in a manner that these resource could optimally be presented, consumed, and converted to other layers of resources for further presentation and consumption.

REFERENCES

[1] David D. Clark. The design philosophy of the DARPA internet protocols. In *Proceedings of the 1988 ACM SIGCOMM Conference*, pages 106–114, Stanford, CA, August 1988.

[2] J. Saltzer, D. Reed, and D. D. Clark. Jerome h. saltzer and david p. reed and david d. clark. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[3] Data Transport Research Group. Survey of transport protocols other than standard grid TCP. Global Grid Forum, 2005.

[4] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for wireless sensor networks. In *W. Weber, J. Rabbey, and E. Aarts, editors, Ambient Intelligence*, New York, NY, 2004. Springer-Verlag.

[5] Computer Business Review Online. ITU head foresees Internet balkanization, November 2005.

[6] Robert Braden, Ted Faber, and Mark Handley. From protocol stack to protocol heap – role-based architecture. *ACM Computer Communication Review*, 33(1):17–22, January 2003.

[7] Samuel K. Moore. Winner: Multimedia monster. *IEEE Spectrum*, 43(1):20–23, January 2006.

[8] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. *SIGCOMM Comput. Commun. Rev.*, 20(4):200–208, 1990.

[9] D. Clark, L. Chapin, V. Cerf, R. Braden, and R. Hobby. Towards the future Internet architecture. In *Network Working Group D. Clark Request for Comments: 1287*, Dec 1991.

[10] David D. Clark *et al.* Newarch project: Future-generation Internet architecture. `http://www.isi.edu/newarch/`.

[11] `http://conferences.sigcomm.org/sigcomm/2004/fdna.html`.

[12] NSF NeTS FIND initiative. `http://www.nets-find.net/`.

[13] `http://www.geni.net/`.

[14] `http://4ward.tssg.org/`.

[15] `http://www.future-internet.eu/`.

[16] Georgios Tselentis, Alex Galis, Anastasius Gavras, Srdjan Krco, Volkmar Lotz, Elena Simperl, Burkhard Stiller, and Theodore Zahariadis. *Towards the Future Internet - Emerging Trends from European Research.* IOS Press, 2010.

[17] M. Zitterbart, B. Stiller, and A.N. Tantawy. A model for flexible high-performance communication subsystems. *Selected Areas in Communications, IEEE Journal on*, 11(4):507–518, May 1993.

[18] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[19] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[20] Nina T. Bhatti and Richard D. Schlichting. A system for constructing configurable high-level protocols. In *Proceedings of the 1995 ACM SIGCOMM Conference*, pages 138–150, Cambridge, MA, August 1995.

[21] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, 1998.

[22] Yu-Shun Wang Joseph D. Touch and Venkata Pingali. A recursive network architecture. Technical report, Information Sciences Instituion at University of South California, Marina del Rey, CA, October 2006.

[23] `http://dedis.cs.yale.edu/2009/tng/`.

[24] Bryan Ford and Janardhan Iyengar. An efficient cross-layer negotiation protocol. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets-VIII)*. ACM, October 2009.

[25] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: Defining tomorrow's Internet. *SIGCOMM Comput. Commun. Rev.*, 32(4):347–356, 2002.

[26] R. Morris and F. Kaashoek. User information architecture.

[27] R. Kahn, C. Abdallah, H. Jerez, G. Heileman, and W. W. Shu. Transient network architecture.

[28] B. Bhattacharjee, K. Calvert, J. Griffioen, N. Spring, and J. Sterbenz. Postmodern internetwork architecture.

[29] D. Krioukov, K.C. Claffy, and K. Fall. Greedy routing on hidden metric spaces as a foundation of scalable routing architectures without topology updates.

[30] D. Duchamp. Session layer management of network intermediaries.

[31] K. Sollins and J. Wroclawski. Model-based diagnosis in the knowledge plane.

[32] M. Allman, V. Paxson, K. Christensen, and B. Nordman. Architectural support for selectively-connected end systems: Enabling an energy-efficient future Internet.

[33] D. Blumenthal, J. Bowers, N. McKewon, and B. Mukherjee. Dynamic optical circuit switched (docs) networks for future large scale dynamic networking environments.

[34] Constantine Dovrolis. What would Darwin think about clean-slate architectures? *SIGCOMM Comput. Commun. Rev.*, 38(1):29–34, 2008.

[35] `http://www.live555.com/`.

[36] `http://www.w3.org/TR/rdf-primer/`.

[37] `http://protege.stanford.edu/`.

[38] `http://www.w3.org/TR/rdf-schema/`.

[39] Manoj Vellala, Anjing Wang, George N. Rouskas, Rudra Dutta, Ilia Baldine, and Dan Stevenson. A composition algorithm for the silo cross-layer optimization service architecture. In *Proceedings of the Advanced Networks and Telecommunications Systems Conference (ANTS 2007)*, Mumbai, India, December 17-18 2007.

[40] Manoj Vellala. Stack composition for silo architecture. Master's thesis, North Carolina State University, Raleigh, NC, 2008.

[41] The SILO NSF FIND project Web site. `http://www.net-silos.net/`, 2006-2010.

[42] Chowdhury Mosharaf and Boutaba Raouf. A survey of network virtualization. Technical report, University of Waterloo, Ontario, Canada, Oct 2008. University of Waterloo Technical Report CS-2008-25.

[43] Chowdhury Mosharaf and Boutaba Raouf. Dynamic internet overlay deployment and management using the x-bone. *IEEE Communications Magazine*, 47(7):20–26, Jul 2009.

[44] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[45] VMWare. Vmware virtual networking concepts. `http://www.vmware.com/files/pdf/virtual_networking_concepts.pdf`, Jul 2007.

[46] Sunay Tripathi, Nicolas Droux, Thirumalai Srinivasan, and Kais Belgaied. Crossbow: From hardware virtualized nics to virtualized networks. In *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 53–62, New York, NY, USA, 2009. ACM.

[47] Sun. Project Crossbow: Network virtualization and resource management. `http://www.opensolaris.com/use/ProjectCrossbow.pdf`, May 2009.

[48] Cisco. Cisco VN-Link: Virtualization-aware networking. `http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns892/ns894/white_paper_c11-525307_ps9902_Products_White_Paper.html`, 2009.

[49] Wikipidia. List of router or firewall distributions. http://en.wikipedia.org/wiki/List_of_router_or_firewall_distributions.

[50] Vyatta. Why Vyatta is better than Cisco. http://www.vyatta.com/downloads/whitepapers/Vyatta_Better_than_Cisco.pdf, 2007.

[51] Virtualization in the core of the network. http://www.juniper.net/us/en/local/pdf/whitepapers/2000299-en.pdf.

[52] Infinera. Bandwidth virtualization enables a programmable optical network. http://www.infinera.com/pdfs/whitepapers/White_Paper_Bandwidth_Virtualization.pdf, 2007.

[53] RFC 4847, 5195, 5251, 5252, 5253, 5523.

[54] Moreno Victor and Reddy Kumar. Transport virtualization - vns. In *Network Virtualization*, chapter 3, page 41. Cisco Press, July 26, 2006.

[55] Cisco. Virtualization beyond the data center. http://www.cisco.com/en/US/prod/collateral/switches/ps5718/ps708/white_paper_c11-531009.html, 2009.

[56] Paul Ferguson and Geoff Huston. What is a VPN? - Part I. *The Internet Protocol Journal*, 1(1), 1998.

[57] Tan Nam-Kee. VPN-in-brief. In *Building VPNs: with IPSec and MPLS*, chapter 1, pages 9–11. McGraw-Hill Professional, July 28, 2003.

[58] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, 2003.

[59] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[60] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.

[61] Paul Brett, Mic Bowman, Jeff Sedayao, Robert Adams, Rob Knauerhause, and Aaron Klingaman. Securing the planetlab distributed testbed: How to manage security in an environment with no firewalls, with all users having root, and no direct physical control of any system. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 195–202, Berkeley, CA, USA, 2004. USENIX Association.

[62] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In vini veritas: Realistic and controlled network experimentation. In *ACM SIGCOMM Computer Communication Review*, Pisa, Italy, October 2006.

[63] Sapan Bhatia, Murtaza Motiwala, Wolfgang Muhlbauer, Yogesh Mundada, Vytautas Valancius, Andy Bavier, Nick Feamster, Larry Peterson, and Jennifer Rexford. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In *CONEXT '08: Proceedings of the 2008 ACM CoNEXT Conference*, pages 1–6, New York, NY, USA, 2008. ACM.

[64] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[65] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 113–128, Berkeley, CA, USA, 2008. USENIX Association.

[66] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, 1997.

[67] Luigi Rizzo. Dummynet and forward error correction. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 31–31, Berkeley, CA, USA, 1998. USENIX Association.

[68] Kevin Fall. Network emulation in the vint/ns simulator. In *ISCC '99: Proceedings of the The Fourth IEEE Symposium on Computers and Communications*, page 244, Washington, DC, USA, 1999. IEEE Computer Society.

[69] Jiang Xuxian and Xu Dongyan. Violin: Virtual internetworking on overlay infrastructure. Technical report, Purdue University, 2003.

[70] Ruth Paul, Jiang Xuxian, Xu Dongyang, and Goasguen Sebastien. Virtual distributed environments in a shared infrastructure. *Computer*, 38(5):63–69, May 2005.

[71] Touch Joe and Hotz Steve. The X-Bone. *Third Global Internet Mini-Conference at Globecom*, pages 59–68, November 1998.

[72] Touch Joe. Dynamic Internet overlay deployment and management using the x-bone. *International Conference on Network Protocols*, pages 59–68, 2000.

[73] `http://www.isi.edu/xbone/`.

[74] E. Grasa, G. Junyent, S. Figuerola, A. Lopez, and M. Savoie. UCLPv2: A network virtualization framework built on Web services [Web services in telecommunications, Part II]. *Communications Magazine, IEEE*, 46(3):126–134, March 2008.

[75] J. D. Touch, Y. S. Wang, L. Eggert, and G. G. Finn. A virtual Internet architecture. Technical report, USC/Information Sciences Institute, Mar 2003. ISI Technical Report ISI-TR-2003-570.

[76] Active networking. http://en.wikipedia.org/wiki/Active_Network.

[77] Stephen F Bush. Introduction to active networks. Video Lecture. http://videolectures.net/contrib07_bush_ian/.

[78] David Tennenhouse and David J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26:5–18, 1996.

[79] Dragos Niculescu. Survey of active network research. http://www.research.rutgers.edu/~dnicules/research/other/active_survey.pdf, July 14, 1999.

[80] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35:80–86, 1997.

[81] Stephen F Bush. GE's active network home. http://www.research.ge.com/~bushsf/AVNMP.html#Anchor-Publications-35882.

[82] Stephen F. Bush and Amit B. Kulkarni. *Active Networks and Active Network Management: A Proactive Management Framework (Network and Systems Management)*. Springer, May 31, 2001.

[83] Research @ run. http://www-run.montefiore.ulg.ac.be/Research/Topics/index.php?topic=Active.

[84] Stefaniab, Michael Devore, and etc. Network virtualization. http://en.wikipedia.org/wiki/Network_virtualization.

[85] William von Hagen. *Professional Xen Virtualization*. Wrox, 2008.

[86] What is network virtualization? http://searchservervirtualization.techtarget.com/sDefinition/0,,sid94_gci1035141,00.html, 09 2008.

[87] http://www.opnet.com/.

[88] http://nsnam.isi.edu/nsnam/index.php/User_Information.

[89] http://www.nsnam.org/.

[90] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[91] http://snad.ncsl.nist.gov/nistnet/.

[92] Mark Carson and Darrin Santay. NIST net: A linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.

[93] Rahul Mangharam. Mixed reality, now a reality: Network virtualization for real-time automotive-cps networks. Technical report, Department of Electrical & Systems Engineering, University of Pennsylvania, Mar 2008.

[94] Nick Feamster, Lixin Gao, and Jennifer Rexford. How to lease the Internet in your spare time. *SIGCOMM Comput. Commun. Rev.*, 37(1):61–64, 2007.

[95] Rudra Dutta, George N. Rouskas, Ilia Baldine, Arnold Bragg, and Dan Stevenson. The SILO architecture for services integration, control, and optimization for the future internet. In *Proceedings of IEEE ICC 2007*, Glasgow, Scotland, June 2007.

[96] Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the Internet impasse through virtualization. *Computer*, 38(4):34–41, 2005.

[97] Elio Salvadori and Roberto Doriguzzi. Network virtualization: A path towards Internet innovation? `http://www.icst.org/portals/content/network-virtualization-a-path-towards-internet-innovation`.

[98] `http://4ward.tssg.org/`.

[99] `http://nicl.cod.cs.duke.edu/orca/index.html`.

[100] Jeff Chase, Laura Grit, David Irwin, Varun Marupadi, Piyush Shivam, and Aydan Yumerefendi. Beyond virtual data centers: Toward an open resource control architecture. In *Selected Papers from the International Conference on the Virtual Computing Initiative (ACM Digital Library)*. ACM, May 2007.

[101] `http://groups.geni.net/geni/wiki/ORBIT/`.

[102] Mohan Iyer. *Providing Bandwidth on Demand Services using Optical Network Design and the SILO Network Architecture*. PhD thesis, North Carolina State University, 2010.

[103] `http://groups.geni.net/geni/wiki/Gec7Agenda`.

[104] Jeff Chase. ORCA control framework architecture and internals. Technical report, Duke University, 2009.

[105] `https://geni-orca.renci.org/`.

[106] `http://www.eucalyptus.com/`.

[107] Extensible messaging and presence protocol (XMPP): Core.

[108] Extensible messaging and presence protocol (XMPP): Instant messaging and presence.

[109] XEP-0060: Publish subscribe. `http://xmpp.org/extensions/xep-0060.html`.

[110] Openfire XMPP server. `http://www.igniterealtime.org/projects/openfire/`.

[111] Smack API. `http://www.igniterealtime.org/projects/smack/`.

[112] Gloox 1.0, XMPP client library. `http://camaya.net/gloox/`.

[113] `http://xmlrpc-c.sourceforge.net/`.

[114] Jeffrey C. Mogul, Praveen Yalagandula, Jean Tourrilhes, Rick McGeer, Sujata Banerjee, Tim Connors, and Puneet Sharma. Api design challenges for open router platforms on proprietary hardware. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, October 2008.

[115] Mitchell Ashley. Network convergence: The unified network platform. `http://www.stillsecure.com/docs/StillSecure_NetworkConvergence_UNP_whitepaper.pdf`, 2007.

[116] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[117] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1995.

[118] Ilia Baldine, Manoj Vellala, Anjing Wang, George N. Rouskas, Rudra Dutta, and Dan Stevenson. Unified software architecture to enable cross-layer design in the future Internet. In *Proceedings of IEEE ICCCN 2007*, pages 26–32, Honolulu, Hawaii, August 2007.

APPENDICES

Appendix A

The SILO User's Guide

Anjing Wang	Ilia Baldine
North Carolina State University	Renaissance Computing
Computer Science	Institute
awang@ncsu.edu	ibaldin@renci.org

A.1 Change Log

v0.3 - More SILO service examples and a SILO GUI Application example

v0.2 - Enabled python-based SCA and the Tuning Agent framework

v0.1 - Basic version of the framework with no Tuning Agent and no SCA

A.2 Introduction

This document describes the architecture of the prototype software demonstrating the capabilities of the SILO concept. The prototype is built in accordance with the Pilot System principle[117], which implies a limited lifespan of the first prototype system. It is intended to be a tool to help us learn about how to properly implement a fully functional SILO framework, as well as how not to implement such a system. The anticipated lifespan of this prototype is 3 years.

A.3 Software Architecture

The architecture will consist of the following major components:

- The SILO framework API

- The SILO ontology
- The SILO-Enabled Application (APP)
- The SILO Composition Agent (SCA)
- The SILO Management Agent (SMA)
- The SILO Tuning Agent (STA)
- Universe of Services Storage (USS)
- Control Strategies Storage (CSS)

This section describes each component in detail and presents the interconnected view of the architecture.

The application communicates with SMA over an IPC mechanism. Initially, it creates a service request, which describes its communications requirements. The request is passed on to the SCA, which constructs a silo recipe. The recipe is then passed to the SMA. The SMA dynamically links in the necessary code and instantiates the state for the new silo using the silo recipe. The application and the SMA communicate by referencing a silo handle. The SMA maintains the silo state. The STA, when necessary, manipulates the control interfaces in order to optimize performance. A control strategy is used to govern the manipulation of the control interfaces. The SMA selects appropriate control strategies from the Control Strategies Storage based on the desired optimization goals. The high-level behavior of the framework is affected by the policies that are currently in effect (as selected by the user and/or system and/or network administrator). The described interactions are shown in Figure A.1.

The combination of USS and CSS is also referred to as SILO ontology. Please refer to [118] for additional details. You may download this paper from `http://net-silos.net/joomla/index.php?option=com_docman&task=doc_view&gid=100&Itemid=39`.

Services are implemented as separate, dynamically loadable libraries, which the SMA loads on demand based on the contents of the silo recipe.

A.4 Download

You may download the current release from `http://net-silos.net`. Please go the Web site, then click SOURCE CODE and you could find the link for downloading: `http://www.net-silos.net/source/SILO_v0.3.tar.bz2`.

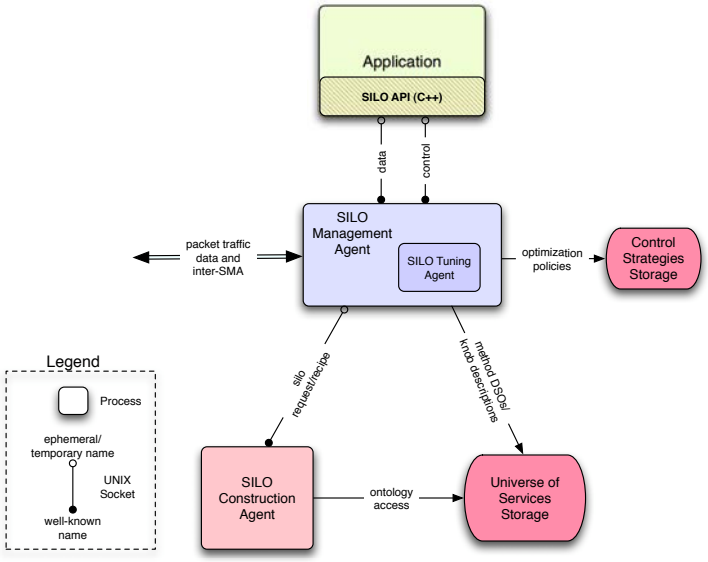

Figure A.1: High-level Prototype Architecture

A.5 Implementation Approach

The SILO prototype is implemented as a series of user-space components implemented in C++, interconnected using traditional UNIX IPC mechanisms (e.g., UNIX sockets, shared memory, message queues, etc.). Due to the need to develop this prototype rapidly, the user-space approach was chosen over a kernel-based implementation. High performance, generally ascribed to kernel-based implementations, is not a high priority in this case. The user-space approach allows us to incorporate the code and components from other OSS projects without regard to their implementation details. It allows us to mix and match implementation frameworks and languages to achieve the fastest result.

The SILO Composition Agent is written in Python, while the rest of the framework is implemented using C++.

A.6 Directory Layout

The silo release includes the following components, each with its own subdirectory.

siloManager/ This directory contains all source code related to SMA (Silo Management Agent).

siloServices/ Several sample services are provided as examples for end users to develop more

services. Each service has a .hh and a .cpp file (example: udp.hh and udp.cpp implementing UDP-like service).

siloManager/test/ Test application is given to test the SMA and services.

siloTuning/ This directory contains all source code related to the STA (Silo Tuning Agent), including the agent code and tuning algorithm implementations.

siloConstructor/ This directory contains all source code related to the SCA (Silo Composition Agent).

ontology/ SILO ontology. The user-viewable ontology should be edited using the free Protegê tool (`http://protege.stanford.edu`).

siloAPI/ Socket-like API for silo-enabled applications. In the test directory, examples of using silo API are provided.

siloAPP/codeblocks/siloGUIDemo A SILO demo with GUI (wxWidgets) developed using Code::Blocks.

externLibs/ Third party libraries to support our code, live555, loki:

live555: http://www.live555.com/

Live555 library is provided in a modified form to include support for UNIX sockets.

loki: http://loki-lib.sourceforge.net/

Loki library is unmodified except for fixing the compling issues in 64 bit OS.

gloox-1.0: http://camaya.net/download/gloox-1.0.tar.bz2

Gloox library is NOT a neccessary part of the SILO framework. It is used to support the pubsub service. It is provided unmodified and for convenience only.

util/ Contains example scripts, recipes and test programs demonstrating the SILO framework capabilities.

A.7 How to Build the Code on Linux

Please follow the steps below to build the SILO framework. Root privileges are not required, except when installing several of the prerequisite components.

The SILO framework is officially supported in Ubuntu (version: 10.04 LTS/9.10/9.04 32/64-bit with gcc-4.4.1 or higher) and Fedora (version: Core12 32/64-bit with gcc-4.4.1 or higher). AS SILO runs in user space, it should be able to be installed in various other Linux distributions as well.

1. Download and install the SILO code in the user acccount. Set the SILOHOME environment variable to the root of the SILO tree:
   export SILOHOME='pwd'

2. We use scons instead of Makefile to build the system. Use the following commands to install scons (*root privileges required, if installing system-wide*).

   yum install scons // Fedora

   apt-get install scons // Ubuntu

   You may also choose to download it from `http://www.scons.org`.

3. Install Xerces XML C++ parser. The SILO framework uses this library as its XML parser. Please download it from `http://xerces.apache.org/xerces-c/`, and follow the instructiosns at `http://xerces.apache.org/xerces-c/install-3.html` for installation.

   We recommend installing the binary distributions from `http://www.trieuvan.com/apache/xerces/c/3/binaries/xerces-c-3.1.0-x86-linux-gcc-3.4.tar.gz` for 32 bit OS, and from `http://www.trieuvan.com/apache/xerces/c/3/binaries/xerces-c-3.1.0-x86_64-linux-gcc-3.4.tar.gz` for 64 bit OS.

   If these links are broken, please download the latest binary distribution; extract the package to <xerces-path>, which could be anywhere on the system (we recommend using /opt/xerces as <xerces-path>); permanently add <xerces-path>/bin to the PATH environment variable; and permanently add <xerces-path>/lib to the LD_LIBRARY_PATH environment variable[1].

   If you use bash, please refer to the following steps.

   tar xvzf xerces-c-3.1.0-x86-linux-gcc-3.4.tar.gz // 32 bit OS

   sudo mv xerces-c-3.1.0-x86-linux-gcc-3.4 /opt/xerces // 32 bit OS

   tar xvzf xerces-c-3.1.0-x86_64-linux-gcc-3.4.tar.gz // 64 bit OS

   sudo mv xerces-c-3.1.0-x86_64-linux-gcc-3.4 /opt/xerces // 64 bit OS

   sudo vim /etc/bash.bashrc // Ubuntu

   sudo vim /etc/bash.rc // Fedora

   Add the following four lines at the end of the bash configruation file

   ```
   PATH=$PATH:/opt/xerces/bin
   export PATH
   ```

[1]Please be careful when you try to edit or add LD_LIBRARY_PATH, as some system may have specific ways to add or edit this variable to avoid potential security issues.

LD_LIBRARY_PATH=/opt/xerces/lib
export LD_LIBRARY_PATH

4. Install libpcap, which is required to compile fauxSCA.

   As you need both library and header files, we recommend downloading from `http://www.tcpdump.org/release/libpcap-1.1.1.tar.gz`. Extract the package first; follow the instructions from INSTALL.txt for three steps: configure, make, and make install (*root privileges required if installing system-wide*). You must ensure that you have necessary libraries in your system to compile libpcap.

   Please use the following commands.

   sudo apt-get install flex byacc bison // Ubuntu

   sudo yum install flex byacc bison // Fedora

   tar xvzf libpcap-1.1.1.tar.gz

   cd libpcap-1.1.1

   ./configure

   make

   sudo make install

   If you encounter problems compiling some of the code with reports about undefined pcap_dump_flush(), remove the *-DHAVE_PCAP_DUMP_FLUSH* flag from the CXXFLAGS option (NOTE: There must be a trailing space left in the CXXFLAGS string)

5. Install rdflib, which is required to run fauxSCA.

   You may choose to download and install rdflib from `http://rdflib.net` (*root privileges required, if installing system-wide*). rdflib requires Python 2.4 or above; source code and headers are required to compile rdflib.

   Please use the following commands.

   sudo apt-get install python-setuptools python-dev // Ubuntu

   sudo yum install python-setuptools-develv // Fedora

   sudo easy_install -U "rdflib==2.4.2"

6. Edit $SILOHOME/SConstruct (scons equivalent of the Makefile) to change XERCES-PATH to <xerces-path>. If you use the recommended path /opt/xerces for <xerces-path>, you may skip this step.

7. Go to directory $SILOHOME/externLibs/live555 and make the live555 library:

cd $SILOHOME/externLibs/live555

make

8. Go to directory $SILOHOME/externLibs/loki and make the loki library:

   cd $SILOHOME/externLibs/loki

   make

   sudo make install

9. Go to directory $SILOHOME/externLibs/gloox-1.0 and make the gloox library if you would like to support the pubsub service:

   cd $SILOHOME/externLibs/gloox-1.0

   ./configure –prefix=/usr –without-examples –without-tests

   make

   sudo make install

10. In $SILOHOME directory, issue the command 'scons' to build the SILO framework. [2].

    If you encounter problems compiling some of the code with reports about undefined pcap_dump_flush() this time, please make sure you are using the right version of libpcap. Uninstall any system libpcap and libpcap-dev package, and delete pcap.h and libpcap.a. Go to the source directory of libpcap-0.9.8, and then issue 'make install again'. In the $SILOHOME directory, use 'scons -c' to clean the build, then use 'scons' to rebuild the project.

A.8 Source Code License

SILO Prototype	GPLv2	`http://www.opensource.org/licenses/gpl-2.0.php`
Loki	MIT-license	`http://www.opensource.org/licenses/mit-license.php`
Live555	LGPL	`http://www.opensource.org/licenses/lgpl-2.1.php`
Xerces C++ parser	Apache 2.0	`http://www.opensource.org/licenses/apache2.0.php`
libpcap	BSD	`http://www.opensource.org/licenses/bsd-license.php`
rdflib	BSD	`http://www.opensource.org/licenses/bsd-license.php`
gloox	GPLv2	`http://www.opensource.org/licenses/gpl-2.0.php`

[2] *scons* includes the functionality of both make and automake/autoconf to enable cross-platform compilations. To speed up compilation, scons supports -j [number] option for parallel builds.

A.9 Configuration and Testing

A.9.1 SILO Universe

The distribution comes with a silo universe file that reflects the available services and tuning algorithms. The universe file is XML-formatted and indicates where the dynamic library for each service/method and algorithm can be located and what their default parameters may be. The universe file included in the distribution uses relative path names and should not require editing. The silo universe used with this release is located in $SILOHOME/siloManager/silo_-universe.xml.

A.9.2 Example Recipes

Several examples of silo recipes are provided in $SILOHOME/siloManager/test/ name silo_-recipe_*. The most complete example is provided in silo_recipe_ip_udp_lb.xml, which is used to create a silo with IP and UDP header services and a leaky bucket service for rate control.

```
<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE silou:UNIVERSE [
       <!ENTITY silou 'http://www.net-silos.net/universe/'>
       <!ENTITY silo 'http://www.net-silos.net/silo/#'>
]>

<RECIPE xmlns="&silou;" xmlns:silo="&silo;" id="1234">

<method name="&silo;leakybucket">
<parameters>
  <para name="size" type="int" value="500"/>
  <para name="rate" type="int" value="100"/>
</parameters>
</method>
<method name="&silo;udp"/>
<method name="&silo;ip"/>

</RECIPE>
```

Normally the SCA takes care of creating the recipes, but some of the low-level testing tools that are not capable of interfacing with the SCA require ready-made recipes.

A.9.3 Testing the Prototype

To test the prototype, please go to the directory $SILOHOME/util/, and then modify *settings* to fit your environment. Specifically, edit SILO_DIR and XERCES_DIR to the paths where the SILO framework is located and where the Xerces C++ parser is installed.

These scripts (respectively) start an SMA in server/client mode, an SCA, a server application, and a client application[3]. The client/server relationship between applications has to do with whether a silo on one end is in server mode or not and internal to the SILO framework. It is similar to server/client relationships when using sockets and is reflected in the SILO API.

Once this is done, you can start the two instances of the Silo Manager and two trivial applications attached to them by first issuing 'sh startServer.sh', followed by 'sh startClient.sh'. Each script will open 3 xterms: one for the SILO Manager Agent, one for the SILO Construction Agent, and one for the application. If everything functions properly, you will observe the client-side application sending messages to the server-side application. If a rate-controlled service, such as the token bucket, is used, only a fraction of sent messages will be delivered.

This version of the code ships with a simple tuning algorithm that optimizes the behavior of the token bucket rate limiter on the client side.

If you have problems when you run startServer.sh, please edit the 15th line of the file to change python to python2.4. Do the same to startClient.sh.

A.9.4 SMA and Services Data Path Testing Tool

This testing tool is located in $SILOHOME/siloManager/test/sdptest, which can build a silo defined by silo_universe.xml and silo_recipe.xml in the same directory. You may test the silo, all the services and the tuning agent by sending data either from the top of the silo (Tx path) or from the bottom to top (Rx path).

1. Type ./sdptest -h for help.
2. If you would like to tune on the tuning agent, use ./sptest -t with other options.
3. Revise silo_universe.xml and silo_recipe.xml as necessary.
4. Use ./sdptest and any other options for the test; if no option is given, the test tool will send a built-in string, "This is test data," from the top of the silo to the bottom and save the output to a file named 'output'.
5. Use wireshark to view the file 'output' containing packet capture data in pcap format.

[3]It is a coincidence that a server application is started on the 'server' SMA and a client application on the 'client' SMA – the two concepts have nothing to do with each other. When we talk about the server/client SMA, we refer to the master/slave arrangement between two communicating SMAs imposed by UNIX sockets. This arrangement becomes meaningless when using raw IP.

A.10 Additional Resources to Explore

SILO Installation and Development Guide details a pre-configured Silo development environment based on Fedora 8, and in the form of a Qemu Virtual Machine image. More details about programming SILO services and SILO tuning algorithms are also elaborated.

SILO API in $SILOHOME/siloAPI/. Test applications (server and client) are in $SILOHOME/siloAPI/test/test_server.cpp and $SILOHOME/siloAPI/test/test_client.cpp, respectively. Currently, a very simple API is supported, allowing the programmer to create a silo request, pass it to SMA and communicate over a created silo using a read/write/accept set of calls.

Service API in $SILOHOME/siloServices. To see examples of how SILO services can be implemented, refer to ip.[cpp, hh], udp.[cpp, hh], tokenbucket.[cpp, hh]. They provide a fairly good view of how new services can be added into the framework.

A.11 Appendix A: Terminology

SILO a framework for creating flexible networking applications.

silo state storage, associated executable code and execution contexts necessary to perform communication functions on behalf of an application. A silo represents a collection of services and methods operating on a data flow.

silo state a storage abstract that maintains information necessary for silo operation (for example, congestion window size, number of packets/bytes transmitted, etc.)

silo handle unique identifier of silo state used between the application and the SILO framework.

service request description of desired services communicated by the application to the SILO framework.

silo recipe an XML-based description of the composition and state necessary to create a silo. Contains pointers to dynamically linkable code to methods constituting a silo.

control interface an abstract describing control options of a specific method within a silo. Control interface is composed of method-specific and service-specific control knobs. Service-specific knobs are inherited based on polymorphism of services and methods.

control strategy an algorithm used to manipulate silo control interfaces in concert in order to achieve a specific optimization goal.

Appendix B

The SILO LAB

This appendix presents part of the achieved Web site used for administrating "ECE/CSC 570 - Computer Networks Fall, 2008, Section 001."

B.1 Info

Location
EB-II 2238 Networking Lab III
Teaching Assistant
Anjing Wang
Time: Monday/Wednesday 2:30 PM - 3:30 PM
Place: MRC-409C
E-Mail: awang AT ncsu DOT edu
Schedule
Lab Calender
Please follow the instructor's suggestions for grouping.

B.2 Registration List

Please refer to Table 5.1.

B.3 Testbed

These desktops shown in Figure B.1 are dedicated to the SILO lab. The full host name of gnl-ipws01 is gnl-ipws01.netlabs.ncsu.edu, and so on. All desktops are installed with REALM linux except for gnl-ipws05; gnl-ipws04 is used for the Internent access; gnl-ipws01, gnl-ipws02,

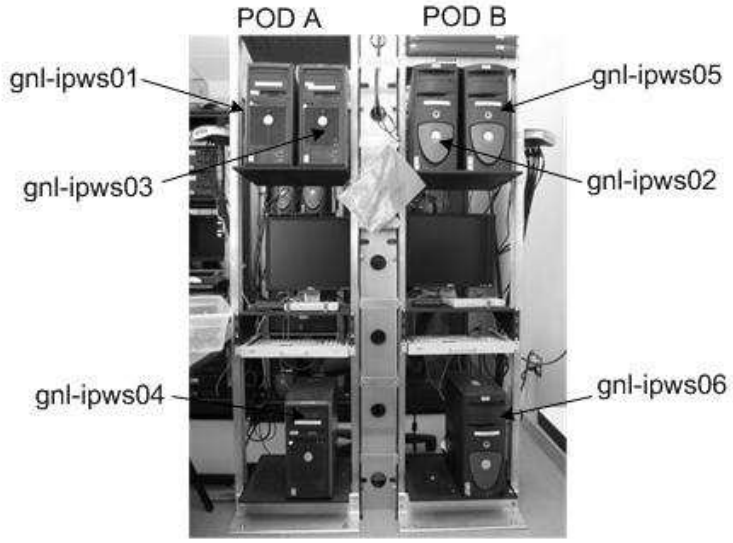

Figure B.1: SILO Testbed

gnl-ipws03 and gnl-ipws06 are installed with the SILO framework and are ready for use by the students.

Three desktops on POD A share the same monitor on the left side, while the other three desktops share the monitor on the right side. Please make sure you switch to the correct screen before you work.

You may need their IP addresses when you run the demo. Please refer to the following table.

Hostname	Public IP	2nd NIC IP
gnl-ipws01.netlabs.ncsu.edu	152.14.83.90	192.168.1.11
gnl-ipws02.netlabs.ncsu.edu	152.14.83.91	192.168.1.12
gnl-ipws03.netlabs.ncsu.edu	152.14.83.92	192.168.1.13
gnl-ipws06.netlabs.ncsu.edu	152.14.83.95	N/A

You may log onto the desktops by a local user, which is your first name. Your initial password will be distributed by your TA. Please make sure to change your password once you obtain the initial password. Then, please work on your $HOME directory, which is /home/yourfirstname. Note: please use the default shell (Bash) to compile and execute the SILO programs, as $PATH and $LD_LIBRARY_PATH are not properly set for other shells.

You may also follow the SILO User's Guide (pdf) to install SILO on your machines. For any installation questions and suggestions for improving User's Guide, please feel free to contact your TA.

B.4 Before You Come to the Lab

1. Go through this Web site.

2. Read this paper: Ilia Baldine, Manoj Vellala, Anjing Wang, George N. Rouskas, Rudra Dutta, Dan Stevenson, "A Unified Software Architecture to Enable Cross-Layer Design in the Future Internet." In **Proceedings of IEEE ICCCN 2007**, August 13-16, 2007, Honolulu, Hawaii. (pdf)

3. Please be aware that the SILO Tuning Agent is disabled for CSC 570.

4. Read the SILO User's Guide (pdf).

5. If you have a linux system, try to follow the user guide and install the SILO framework on your own.

B.5 Step-by-Step Guide for the First Walk-through

1. Log onto one of the silo-enabled desktops (gnl-ipws01, gnl-ipws02, gnl-ipws03 or gnl-ipws06) with your first name and initial password. We call it A.

2. Open a terminal, and change the password by the command 'passwd'. No parameter is needed for 'passwd'.

3. Open this Web site (`http://courses.ncsu.edu/csc570/lec/001/lab.html`), and download the SILO source code (tar.bz2). If you use Firefox to download the source code, it will be placed on your desktop. Please move it into your $HOME directory, which is /home/yourfirstname. Please do not use the source code on the official SILO Web site because the above code was customized for CSC 570.

4. Open a command terminal, go to your $HOME directory, and use command 'tar xvfj silo-src-570.tar.bz2' to extract the source code. Then, you get the entire source code in the directory $HOME/silo-src-570.

5. Go to directory $HOME/silo-src-570/externLibs/live555, and use command 'make' to make the live555 library, which is the event scheduler of the SILO framework.

6. Go to directory $HOME/silo-src-570/externLibs/loki, use command 'make' to make loki, which provides smart pointer and widely used in the SILO framework.

7. Go to directory $HOME/silo-src-570/, and use command 'scons' to build the SILO framework. 'scons' is a tool similar to 'make', and it has been installed on the lab machines.

8. Log onto another silo-enabled desktop B, and redo Steps 1 through 7.

9. Start server. In A, go to directory $HOME/silo-src-570/siloManager/test/, use './sdptest -a <ip-address>' to start server. -a <ip-address> is used to identify the interface on which the server will run. We recommend that you use the second NIC when running SILO, if is a second NIC in the desktop. Please refer to the above IP table for the desktops' IP addresses, or you

may call '/sbin/ifconfig' to view the interfaces. You may also call './sdptest -h' to get help for this test tool.

10. Start Client. In B, go to $HOME/silo-src-570/siloManager/test/, and issue the command './sdptest -d <ip-address-of-server>' in a terminal. You need to use the IP address of the server you just used in Step 9, not the IP address of this client.

11. See the data flow from B to A.

12. Close the client and server by Ctrl-C, and log out of your account.

B.6 After the First Walk-through

1. Kindly let your TA know of any errors or inaccuracies in this Web site.

2. Take a look at the source code, and you may start from the /siloManager/test/testSiloTool.cpp. Understand this test tool is a combination of SMA, SCA and APP, which are described in the software architecture paper.

3. Take a look at *.cpp and *.hh in the directory /siloServices/. Start from blank.hh and blank.cpp. Read the comments in these two files.

4. Come up with ideas about services you would like to design for the CSC 570 project.

5. If you have any questions or would like to discuss your design with the TA or the instructor, feel free to contact us and make an appointment.

B.7 Submission

Lab report: Server design overview, function description, test results, etc.

Source code: The directory of siloServices and other necessary files and directories.

B.8 Official SILO Web site

http://www.net-silos.net/

B.9 Documents & Papers

The SILO User's Guide (pdf)

Rudra Dutta, George N. Rouskas, Ilia Baldine, Arnold Bragg, Dan Stevenson, "The SILO Architecture for Services Integration, controL, and Optimization for the Future Internet." In **Proceedings of IEEE ICC 2007**, June 24-27, 2007, Glasgow, Scotland. (pdf)

Ilia Baldine, Manoj Vellala, Anjing Wang, George N. Rouskas, Rudra Dutta, Dan Stevenson, "A Unified Software Architecture to Enable Cross-Layer Design in the Future Internet." In **Proceedings of IEEE ICCCN 2007**, August 13-16, 2007, Honolulu, Hawaii. (pdf)

Manoj Vellala, Anjing Wang, George N. Rouskas, Rudra Dutta, Ilia Baldine, Dan Stevenson, "A Composition Algorithm for the SILO Cross-Layer Optimization Service Architecture." In **Proceedings of the Advanced Networks and Telecommunications Systems Conference (ANTS 2007)**, December 17-18, 2007, Mumbai, India. (pdf)

Last updated: Nov-03-2008 11:15am