Infrastructure Support for Accessing Network Services in Dynamic Network Environments

by

Xiaodong Fu

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 2003

Approved: _____

Research Advisor: Vijay Karamcheti

To my wife, Qinghua

Acknowledgment

First and foremost, I want to thank my advisor: Vijay Karamcheti for his guidance over these years. It is from him that I learned how to do research, present ideas, and a lot more. I benefited tremendously from his profound insight and passion for research work that really matters. He is the advisor who is always willing to listen, discuss, and help, not only on research problems. I had the great fortune of working with you, Vijay!

I owe a great deal to Professor Zvi M. Kedem for his advice, insight and support during these years. I thank Professor Partha Dasgupta at Arizona State University for his help in my project during his stay in NYU. I would also like to thank Professors Allan Gotllieb, Benjamin F. Goldberg and Michael L. Overton for kindly serving on my dissertation committee and providing valuable input for improving this thesis.

I also want to thank all members of the parallel and distributed system group. It is a great working environment. In particular, I want to thank Weisong Shi, Anatoly Akkerman, Fangzhe Chang, TaoZhao, Hua Wang, and Eric Freudenthal for their help in my projects, I also benefited a lot from discussion with Anca-Andreea Ivan, Congchun He, Feng Tang, Tatiana Kichkaylo, Yuanyuan Zhao, Kazumune Masaki, and Xin Yu. And finally, I am thankful for the time and fun we had together.

I dedicate this work to my wife, Qinghua Liu, for her love, encouragement and support throughout the sometimes arduous and stressful period of being a graduate

student. Without her, this is not possible. I also want to thank my parents, Daiwen Fu and Wanmei Liu, for their constant love and support throughout my life.

This work was sponsored by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929; by NSF grant CAREER: CCR-9876128 and CCR-9988176; and Microsoft.

Abstract

Despite increases in network bandwidth, accessing network services across a wide area network still remains a challenging task. The difficulty mainly comes from the heterogeneous and constantly changing network environment, which usually causes undesirable user experience for network-oblivious applications.

A promising approach to address this is to provide network awareness in communication paths. While several such path-based infrastructures have been proposed, the network awareness provided by them is rather limited. Many challenging problems remain, in particular: (1) how to automatically create effective network paths whose performance is optimized for encountered network conditions, (2) how to dynamically reconfigure such paths when network conditions change, and (3) how to manage and distribute network resources among different paths and between different network regions. Furthermore, there is poor understanding of the benefits of using the path-based approach over other alternatives.

This dissertation describes solutions for these problems, built into a programmable network infrastructure called Composable Adaptive Network Services (CANS). The CANS infrastructure provides applications with network-aware communication paths that are automatically created and dynamically modified. CANS highlights four key

mechanisms: (1) a high-level integrated type-based specification of components and network resources; (2) automatic path creation strategies; (3) system support for low-overhead path reconfiguration; and (4) distributed strategies for managing and allocating network resources.

We evaluate these mechanisms using experiments with typical applications running in the CANS infrastructure, and extensive simulation of a large scale network topology to compare with other alternatives. Experimental results validate the effectiveness of our approach, verifying that (1) the path-based approach provides the best and the most robust performance under a wide range of network configurations as compared to end-point or proxy-based alternatives; (2) automatic generation of network-aware paths is feasible and provides considerable performance advantages, requiring only minimal input from applications; (3) path reconfiguration strategies ensure continuous adaptation and provide desirable adaptation behaviors by using automatically generated paths; (4) both run-time overhead and reconfiguration time of CANS paths are negligible for most applications; (5) the resource management and allocation strategies allow effective setting up shared resource pools in the network and sharing resources among paths.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

1.1 Motivation

The role of the Internet has undergone a transition from simply being a data repository to one providing access to a large set of sophisticated network-accessible services such as e-mail, banking, on-line shopping, and entertainments.

However, accessing network services across a wide area network still remains a challenging task. This is especially the case as an increasing number of users use portable devices such as PDAs, Pocket/Handheld PCs, cellular phones and two-way pagers with a variety of networking options ranging from Bluetooth [28] to Wireless 3G [54]. Examining a typical communication path between a client application and the visited server (as shown in Figure 1.1), one can observe that the path usually involves multiple links. These links can have very different bandwidth, delay, and error characteristics, ranging from serial links to wireless to broadband to fiber. In addition to these differences in network links, the nodes along the path can also have very dif-

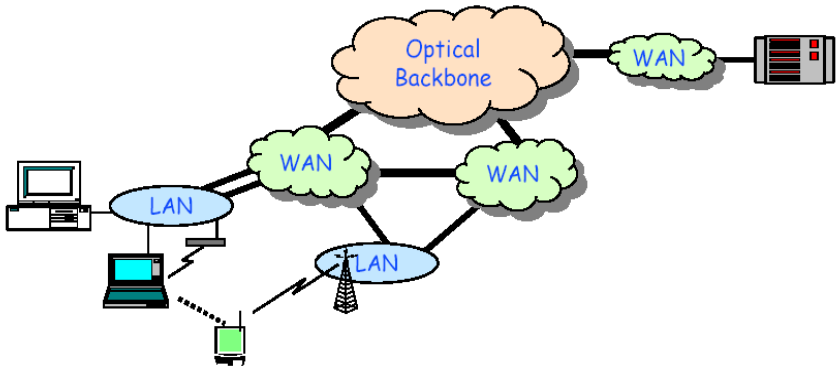


Figure 1.1: Communication paths between clients and Internet services.

ferent capabilities (most true for the end device). Tables 1.1 and 1.2 list the diverse characteristics of some of the links and devices that are currently used in the Internet. Further complicating service access is the fact that the load on the network resources along a communication path may change continually. When running in such heterogeneous and constantly changing environments, applications require quality guarantees in data communication for delivering satisfactory user experiences. For example, a media player may require the communication path to sustain 30 frames per second in order to produce an appropriate display at the end devices.

Nevertheless, the Internet still remains a best-effort platform for delivering data packets. Even with many proposals for providing Quality of Service such as QoS-IP [24], RSVP [26] MPLS-TE [25] etc., provisioning of communication paths with guaranteed QoS is usually expensive and may not be available for many applications. More importantly, the low level QoS parameters may not be able to be mapped directly to application performance requirements (e.g. translating bytes per second to application-specific frames per second). Consequently, the current situation is that

quality of data communication of applications is directly affected by the underlying network conditions, which can result in poor performance or undesirable behavior perceived by the end user unless the application is written to explicitly handle the changes in network conditions.

However, what complicates the construction of such applications is the fact that the communication abstractions provided by traditional transport protocols, instead of exposing network conditions to applications, tries to hide them. Moreover, these abstractions are too high-level (i.e. for all types of applications) for applications to specify their specific requirements, not to mention to allow applications to excise any control over data communication in their preferred fashion. For example, TCP [34] provides applications with the abstraction of an end-to-end reliable byte stream, and it also contains mechanisms for handling flow control and a few exceptional network conditions (e.g. congestion). However, TCP does not allow application to specify how to cope with the condition when the bandwidth of an individual link drops to some level, which causes a decreased throughput at the receiving end. Such changes require very different handling between banking applications and media streaming applications.

The combination of these factors: heterogeneous and dynamic changing network environment and the lack of application specific control over data communication across the network, can cause poor performance or unsatisfactory user experiences for network-oblivious applications.

To improve user experiences while accessing Internet services, a widely adopted solution today relies on differentiated service for different user groups. For example, many popular news, stock trading services, or media streaming services provide

Link	Bandwidth
56K Analog Modem	56 Kbps
Frame Relay	56 Kbps-1.544 Mbps
WiFi	11Mbps
Ethernet (0-1 hops)	100-1000 Mbps
ADSL	1.5 to 8.2 Mbps downstream, 64K–1 Mbps upstream
SDSL	1.544/2.048 Mbps
T-1	1.544 Mbps
E-1 (Europe)	2.048 Mbps
T-3 (or DS3)	44.736 Mbps
E-3 (Europe)	34.368 Mbps
OC-3	155.52 Mbps
OC-12	622.08 Mbps
OC-48	2.488 Gbps

Table 1.1: Bandwidth of Some Links in the Network

System	CPU	Memory
Sun Fire 15k	Up to 106 UltraSPARC III Cu 1.2-GHz Processors	More than 1/2 TB Memory in a single domain. Up to 18 fifth-generation Dynamic System Domains
Dell Dimension 8300	Intel Pentium IV processor 3GHz	Up to 2 GB Dual Channel DDR 400MHz SDRAM
Compaq Evo N410c	Intel Pentium III Mobile 1.2GHz	256MB SDRAM-133 MHz
iPAQ h3955 pocket pc	400MHz Intel XScale processor	64MB RAM, 32MB Flash ROM
Palm m515 handheld	Motorola Dragonball VZ 33 Processor	16MB RAM

Table 1.2: Properties of Some Computer Nodes in the Network

multiple “versions” of the service for different clients. The selection of a suitable version is usually determined by the client connection option. Though this approach can address the heterogeneity problem to some extent (at least for the last hop), it

cannot satisfactorily handle resources for which the availability changes continually, e.g., when the available bandwidth of a network link along a communication path decreases as a result of increased traffic in the network.

1.2 Network Awareness

The problems described above reveal the need for *network awareness* in data communication. The meaning of network awareness here is twofold. First, it means that data communication should be aware of underlying network conditions, which may change dynamically. Second, it means that data communication should also have the knowledge of application performance requirements, which are directly related to the way in which data is interpreted and used by the application. Combining these two together, a network-aware communication path should be able to match application performance requirements with the underlying network resource availability, and further continually adapt to dynamic changes in the network, using the requirements as a guide.

To highlight the benefits of network awareness, let's consider the following example scenario: Alice starts her day by initiating a meeting with one of her clients in another city using a net-meeting application, which runs on a laptop with an IEEE 802.11b wireless connection. During the meeting, some of her colleagues start to download large files using the same office network. Just before Alice notices a long response time in conversation as a result of these download activities, the communication path realizes this problem and automatically starts to drop some less important video frames to maintain a desirable throughput (at the cost of a slightly blurred im-

age). When some of these download tasks are completed, the image quality comes back to normal because the path automatically stops dropping video frames after realizing that there is sufficient bandwidth available once again. As the meeting continues, Alice decides to get her lunch from a public cafe. To continue with her meeting, she hands over the meeting session to a Pocket PC, which she can conveniently carry to that cafe. Realizing that the network in the public cafe is different from the office network, the path automatically encrypts data in transmission and decrypts it upon receipt at her Pocket PC.

This example scenario highlights the benefits from network awareness in data communication. Unlike a traditional data communication path that provides high-level abstractions such as reliable byte streams, a network-aware communication path understands application specific performance requirements and can accordingly change its behavior under different network conditions. Without the support for such network awareness, either applications themselves have to cope with these problems (e.g. changes in link bandwidth or security properties of the network environment in the above example) or the user (Alice) will end up with an unsatisfactory experience (e.g. long response time in conversation or leakage of sensitive data in the above example).

From the perspective of applications, using performance requirements to guide behavior of communication paths under different network conditions allows them to control or customize data communication in the network.

1.3 Possible Approaches

The network awareness described above can be realized in different ways. A widely used approach is to encode all adaptation logic into the client and server applications. An example of such an approach can be found in many commercial media players. When such a media player running on the end device detects that a large number of video frames arrive after their associated deadlines, it notifies the media server to switch to another stream with different data fidelity. After that, the communication path may be able to sustain the required throughput.

Using such an approach requires considerable programming effort. Developing a workable solution requires a comprehensive knowledge of network communication and a deep understanding of how underlying network state affects the performance of the particular application. Besides, such a hard-coded approach is usually hard to extend to new applications.

Compared with this hard-coded approach that targets at a particular application, a more general solution is to provide an adaptation framework that can be used by different types of applications. These frameworks provide necessary support for various applications to cope with different network conditions, thus reducing the direct effort required from applications themselves. Based on where adaptation occurs, these adaptation frameworks can be divided into three groups: end-point approaches, proxy-based approaches and path-based approaches.

In an end-point approach, the client and server cooperate to determine how they should communicate with each other under different network conditions. Many strategies used in the hard-coded server/client approach can also be applied here, with the

underlying infrastructure providing common system support. However, the constraint that adaptation can occur only at end points may limit the adaptation solutions that can be used. For example, some end devices may have insufficient capability to do required computation (e.g. decompression/decryption). In other cases, such a constraint may compromise agility of adaptation: it may take a long time for end points to respond to changes in the network, especially for long communication paths in a wide area network.

For a proxy-based approach, proxy sites along communication paths are exploited to realize the network awareness. These proxies can be used to do transcoding and content distillation for different client classes. This approach relieves servers from the responsibility of having to cope with different network conditions, thus simplifying the task of server construction. Besides, managing a large proxy site that is devoted to handling adaptation may also offer some economic advantages as compared to managing a large set of servers. But the limitation that adaptation occurs only at proxy sites (in most cases just before the last hop), like the end-point approach, again results in similar problems in adaptation agility and limited solution spaces. Moreover, this approach may result in resource wastage along a communication path before the proxy node. For example, if the bandwidth of the last link along the communication path of a media streaming application drops, the proxy will have to drop some media frames to deliver the required throughput. Though such an adaptation solves the low bandwidth problem, considerable bandwidth has already been wasted on the links before the last hop.

Compared with the end-point and the proxy-based approaches, a more general approach is to use all (or as many as possible) network resources along a communi-

cation path for network awareness. We refer to this as a *path-based* approach. While several frameworks have been proposed to improve data communication performance by introducing various components into communication paths, the network awareness achieved using existing path-based infrastructures is rather limited. The reason for this is because that several key questions in the path-based approach still remain unanswered. In particular, for a path-based infrastructure to be really useful, it must contain effective solutions to address the following concerns: how to separate logic that controls data communication from other parts of an application? how to *automatically* construct new paths and modify existing paths so that applications always achieve the best performance for the underlying network conditions? how to manage network resources in a large scale network for such path-based infrastructures? And finally, what are the benefits of using the path-based approach as compared to other alternatives?

1.4 Goals and Approach of This Dissertation

This dissertation present a path-based framework with a complete set of solutions to all of these questions mentioned above. In our approach, a communication path is augmented with application-specific components, which are deployed throughout all (possible) network resources between the server and the client. These components, which can transparently handle stream degradation, reconnection, and in general support arbitrary transcoding, caching, and protocol conversion operations, serve to "impedance match" the application performance requirements with the underlying network conditions. The important thing is such an augmented path is aware of ap-

plication performance requirements and can automatically and continually adapt to changes in the network, thus providing exactly the network awareness we described in Section 1.2.

Our approach has been realized in a general adaptive network infrastructure that provides network-aware paths for applications whose performance is related to the "quality" of underlying data communication. Network-aware paths are automatically created by the underlying infrastructure, requiring only high-level input from applications. Automatically generated paths provide optimized performance to applications by customizing their behaviors to the network conditions encountered at run time. Furthermore, when underlying network conditions change, such paths, both globally and at the level of individual segments, can *continually modify their behaviors according to the performance requirements of the application*. Both path creation and reconfiguration are handled by the underlying infrastructure; therefore, regular (network-oblivious) applications can easily be augmented with network awareness without requiring onerous effort from application developers.

The infrastructure embodies our belief that an appropriate balance between the need for custom data communication and system extensibility is needed. It is based on the observation that on one hand, common high-level abstractions of communication paths usually suffer from poor performance in a dynamic network environment because applications can not specify their requirements; on the other hand, approaches that require the application to take care of every aspect in data communication are usually not extensible. Our infrastructure achieves both of these goals.

- First, it allows custom control over communication paths using various applica-

tion specific components. The selection of components requires only high-level information from the application. Unlike conventional abstractions, these components understand data in transmission, thus can process it in accordance with application performance requirements. This part of our approach provides applications with custom control over data communication in an extensible fashion.

- Second, it separates application "business" logic from what is used for creating and controlling such augmented paths so that application developers only need to concentrate on the former. Once high-level objectives (application performance requirements) are specified by the application, the logic for creating and controlling paths is application-neutral and can be handled by the infrastructure. This part of our approach relieves applications from having to undertake this responsibility, and provides a general way to construct adaptation solutions.

1.5 Contributions

This dissertation explores how to provide network-oblivious applications with network awareness in data communication using a path-based approach. The contributions of this dissertation include the following:

- A *high-level integrated specification* of components and network resources to model behaviors of both components and network resources. This specification allows late binding of components to paths, which is essential for flexibility of dynamic compositions.
- *Automatic path creation strategies* for constructing network-aware access paths for applications. The generated network paths provide optimized performance

in accordance with application performance requirements and underlying network conditions.

  - The path creation strategies can satisfy different performance requirements, i.e. maximize (minimize) value of some performance metric, or guarantee that some performance metric lies in a required range.
  - In addition to constructing an end-to-end communication path, the strategies can also work on disjoint portions of a path independently while maintaining overall performance requirements.
  - The strategies allow augmented paths to be incrementally built across different network domains in a distributed fashion.

- System support for *low-overhead dynamic path reconfiguration*. Path reconfiguration in our infrastructure provides semantic continuity guarantees for data transmission, and is carried out without requiring involvement from applications. Our reconfiguration strategies can be used to modify the entire communication path as well as disjoint portions of the path concurrently and independently.

- *Distributed resource management strategies*, which can be used to manage resources among multiple paths and different network regions so as to improve performance of both individual paths and the whole network.

- An adaptive network architecture called Composable Adaptive Network Services (CANS). CANS is built from the ground up to embody our approach. A series of experiments have been conducted on CANS with different types of applications, the results validate the effectiveness of our approach.

- Extensive performance comparison among end-point, proxy-based, and path-based approaches by simulating their behaviors in a large network topology. Our simulation results show that the path-based approach provides the best and the most stable performance under different network configurations.

1.6 Organization

The rest of this dissertation is organized as follows.

Chapter 2 reviews related work and open questions for path-based infrastructures. Chapter 3 presents the overall architecture of our framework. Chapter 4 describes the type model used for specifying component and network resource behaviors. Chapter 5 describes automatic path creation strategies. Chapter 6 discusses system support for dynamic path reconfiguration. Chapter 7 explains resource management strategies. Chapter 8 describes the implementation of CANS. Chapter 9 shows a comprehensive evaluation of our framework. Chapter 10 concludes this dissertation.

Chapter 2

Background

In this chapter, as high-level context for our framework, we first highlight existing trends that show increased computation and control functionality being introduced into the network. After that, we discuss three groups of related efforts: general mechanisms for introducing control functions into the network, component-based communication systems, and general adaptation frameworks.

2.1 Networking

2.1.1 Changes in the Internet

Despite their tremendous popularity and deep impact on the way people live, computer networks have a relatively short history. It was in the late 1950s and the early 1960s when the first form of networking appeared, and the Internet is only about 34 years old. Nevertheless, several things have changed dramatically in such a short period of time. The changes are mainly in three aspects. The first is the size of the

network. When the ARPANET was first set up in 1969, there were only 4 hosts, but now, there are more than 170 millions hosts connected to the Internet and this number is still growing very fast. Advances in hardware technology are responsible for the second big change: both available bandwidth and connection options have increased dramatically. While it was not a long time ago that most people believed that a 28.8Kbps modem connection would satisfy the data communication needs of all types of applications, now a lot of people have broadband connectivity (with more than 1.5Mbps bandwidth) at home, and continue to feel that the bandwidth is not enough. Pervasively used portable devices and wireless connectivity (Cellular Phone, Bluetooth [28], Wireless 3G [54]) are making the Internet an even more heterogeneous environment. The third change, which in the author's opinion is the most important reason for the success of the Internet, is the growth of available applications. In the early days of the Internet, available applications were limited to four types: email, newsgroups, file transferring, and long distance computing. Now, innumerous applications are running on the Internet: online gaming, shopping, banking, trading, driving directions, real time news and multimedia streaming, to name a few. These applications have pervaded every aspect of people's lives; meanwhile, new applications continue to emerge in the Internet everyday.

2.1.2 Advances in Communication Platforms

On the other hand, the communication platform supported by the Internet has not seen as much improvement. Since 1982, when the TCP/IP protocol (Transmission Control Protocol [34] and Internet Protocol [33]) was established as the standard of the ARPANET, it is still the only reliable delivery service available for all Internet

applications. This naturally leads to the following question: despite the fact that the Internet has changed so much (in network size, hardware technology, and application diversity), is the TCP/IP protocol still sufficient for the data communication needs of all applications?

Basically, IP provides a best effort platform for delivering network packets. The function of a network node in an IP network is merely to forward incoming packets (using a routing table and a fixed routing algorithm). Built on top of the IP layer, TCP supplies applications with the abstraction of a reliable, in-order, unstructured byte stream between two end points. Though such an abstraction was quite sufficient for early Internet applications (email, ftp etc.), people are beginning to realize the limitation of this view as network complexity and application diversity grow. While this simple view has gained TCP/IP tremendous success in that it has been deployed to hundreds of millions of hosts around the world, the same view will eventually bring severe constraints as networks and applications become more complex.

2.1.3 What is Missing?

Several ongoing efforts are investigating extensions of existing protocols as well as proposing new protocols to address these perceived shortcoming. For example, there exist many proposals (QoS-IP [24], RSVP [26] etc.) for providing QoS guarantees in data communication for applications. However, no wide deployment is currently available. The difficulty in deploying such new protocols mainly comes from the size of the Internet and more importantly the network view of the IP network: since the function of a network node is merely to forward packets, deploying new protocols and communication services that require significant changes in the whole infrastructure is

almost impossible (or at least takes a very long time).

In 1994–1995, the concept of active networking emerged [51], which proposed a general mechanisms for extending the functionality of network nodes to support execution of code embedded in network packets: in addition to passive data, a packet in an active network could contain some executable code (or references to code). The network nodes (routers) in an active network are required to execute the accompanying code upon receiving an incoming packet. The code, not limited to just route packets, could perform arbitrary computation on the packets, including modification of the packet itself. An important anticipated use of active networking was for deploying new network protocols over the network. Such an approach can certainly be used to bring applications more control over the data communication in the network, but it entails significant modification of the existing infrastructure.

Recent work on overlay networks ([3] [49] [12] [9]) reflects the same idea that more functionality should be introduced into the network in order for applications to perform better. Realizing the difficulty in modifying the existing infrastructure, overlay networks try to bring in additional functionality on top of the existing infrastructure. For example, a resilient overlay network (RON) [3]) is an application-layer overlay on top of the existing Internet routing substrate where each overlay node monitors the functioning and quality of the Internet paths between itself and other overlay nodes. RON can be used by distributed applications to detect and recover from path failure, often much faster than TCP/IP (within the range of several seconds instead of several minutes when TCP/IP is used). Moreover, it can also improve performance of data communication, i.e. loss rate, latency, or throughput perceived by applications.

In summary, as the Internet exhibits increasingly complex behaviors and the di-

versity of applications increases, the need for custom functionality in the network also grows. From the perspective of applications, this means more control over data communication for applications to obtain better performance; in other words, data communication should be aware of network conditions and application requirements.

2.2 Related Efforts

Realizing that the communication abstractions provided by the Internet, which treat all applications uniformly, usually are inefficient, researchers have been studying enabling mechanisms for applications to customize or control their data communication.[1] The large number of proposed approaches share a common theme: to provide communication paths augmented with specific functionality required by end applications.

In this section, we review previous efforts that are most related to our framework. General network-layer mechanisms for introducing more functionality into the network are discussed first. Then, we briefly describe several communication systems that are constructed from small components. These works show that the component paradigm can be applied to build extensible communication systems, without compromising performance. Finally, we discuss general adaptation frameworks that can improve application performance by providing support for applications to cope with different network conditions. As part of this, we examine the current status of the path-based approach and open problems that need to be addressed before the network awareness described in Chapter 1 can be realized.

[1] A similar phenomenon can be found in works on extensible OS (ExoKernel [17], SPIN [5] etc.).

2.2.1 Network Layer Approaches

Functionality for data communication can be introduced at the network layer.

Transformer tunnels [50] is an approach that allows users to specify functions to transform packets (e.g. compression, encryption) on an individual link (especially for a last-hop link) along a communication path based on its characteristics. For example, if a portable device in a shared wireless network environment needs to preserve the privacy of its data transmission, encryption/decryption functions can be inserted at the both ends of the wireless link. Functions associated with a transformer tunnel need to be configured by the users themselves, thus requiring a comprehensive knowledge of the underlying network substrate. Moreover, since this mechanism works on the network layer, the associated functions of a transformer tunnel will be applied to all communication paths passing through the tunnel, independent of the applications they belong to. This approach is most applicable for the last-hop link, especially for mobile hosts.

With a similar goal but unlike this link-oriented approach, protocol boosters [38] is an end-to-end mechanism that can provide similar functionality using an extensible protocol stack. Using this approach, protocol elements (called protocol boosters) can be transparently inserted into and deleted from the protocol graph on an as-needed basis. For example, if both ends of a communication path have encryption/decryption elements installed, then the application can send and receive data using this ”boosted” secure protocol. Though special protocols like IPSEC [58] can also provide similar functionality, the difference is that those special protocols are hard coded while the protocol booster provides a general framework to extend the functionality of an

existing protocol stack.

More general active network infrastructures, such as ANTS [55], Switchware [1], Netscript [60] NodeOS [23] etc., propose mechanisms on the network layer for packets to carry arbitrary functions in addition to passive data. The accompanying functions are executed in the network nodes (routers) upon the associated data.

For example, in an ANTS [55]-based network, capsules carry references to functions in some protocol. Upon receiving such a capsule, an ANTS node executes the referred function to process the capsule. Moreover, a small amount of soft state can be left behind at the processing node so that the execution of subsequent packets can leverage the information. In ANTS, code is distributed dynamically on-demand, therefore a capsule needs to carry only function references instead of the actual code. Caching mechanisms are further exploited to reduce the overhead associated with transferring code.

In summary, these approaches can be used to introduce additional functionality into communication paths. However, solutions built on the network layer are limited in the following two ways. First, deploying such approaches usually requires significant changes to the existing infrastructure, which is infeasible in most cases for a wide area network. Second, computation conducted on packets usually lack the information of application-level information of the data in transmission and how the application interprets or uses the data. This can considerably limit the performance improvements that can be achieved using such approaches.

2.2.2 Component-Based Communication Systems

The component paradigm (COM [52], Corba [29], JavaBeans [15], EJB [39] etc.) has been successfully used in building large, complex, but extensible software systems. The central idea is to construct complex software systems by composing small components together. These small components have relatively simple functions and well-defined interfaces for interaction amongst them. This paradigm can also be applied to build extensible, component-based communication systems.

X-kernel [32] is such an architecture for constructing communication protocols. In the X-kernel, a complex communication protocol is decomposed into a graph of micro-protocols and virtual protocols. A multistage approach is proposed to decompose complex protocols and construct new communication services. In a protocol graph, each micro-protocol is basically a module with a simple function (e.g. padding the message header out to a pre-determined length); virtual protocols are used to replace selection logic (**IF** statement) in a complex protocol for non-linear composition. For these protocol modules to be composed in arbitrary ways, they have to conform with a set of properties, which is called the meta-protocol.

Similar ideas can also be found in systems like Click [41], Cactus [30], Ensemble [6], and Router Plugins [14] etc. For example, Click [41] is a configurable router, for which the routing function is implemented as a graph of components with a common interface. A component in a Click router is called an element, which has input and output ports that can support push or pull mode operations respectively and can be connected together. Different routing functions can be realized by configuring different element graphs. Ensemble [6] proposes a layer approach to stack micro-protocols

together, each of which handles some small aspect of the required communication guarantees. Cactus [30] further exploits the event paradigm, each micro-protocol is implemented as a collection of event handlers. The primary benefit of using the event paradigms is the indirection implicit in it, which makes micro-protocol binding/unbinding very convenient.

The Scout operating system [42] extends this idea further from communication subsystems to the data flows between operation systems and applications. It uses the data flow as an explicit abstraction in OS design, called *paths*. A path in the Scout operating system is a linear data flow from one device to another (e.g. from a SCSI card to an ETH card). The OS kernel consists of a graph of components (called routers). Paths are created, managed and deleted dynamically for applications. Path creation in Scout involves two steps: first a feasible sequence in the route graph is identified; then the chosen sequence is optimized according to a set of preexisting rules. The path abstraction is very useful, especially for applications whose logic can be embodied as a sequence of data flows across different modules in the local operating system. An example can be an application that encodes MPEG and sends it to the network. The primary benefit of using the path abstraction over traditional process/thread models is that the use of the path abstraction can make scheduling and admission control of OS resources much easier and more stable.

These works demonstrate that the component paradigm can be used to build extensible communication systems. Moreover, component-based communication systems are also feasible from a performance perspective: as the various systems above have demonstrated, components can be used in communication systems without incurring a significant performance penalty as compared to monolithic implementations.

2.2.3 Adaptation Frameworks

The general adaptation frameworks most related to the work in this dissertation focus on a very specific goal: improving application performance by enabling applications to adapt to different network conditions.

Most of these frameworks are built on the application layer, so information about applications and the data in transmission can be exploited to enhance the network awareness achieved. Depending on where adaptation operations occur, these works can be further categorized into three groups: end-point approaches, proxy-based approaches and path-based approaches.

End-Point Approaches

An adaptation framework is called an *end-point approach* (Rover [35], InfoPyramid [40], Odyssey [45] etc.) if it uses only client and server nodes in adaptation. Odyssey [45] is such a general framework that allows client applications to register their expectations of resource availability. The framework is responsible for monitoring resource availability and producing notifications whenever the registered resource expectations can no longer be met. Responding to such notifications, client applications may change data fidelity accordingly. The cooperation protocol for changing data fidelity level is handled by the server and a component on the client side called a Warden. Though such a general framework simplifies construction of adaptation solutions by providing common system support such as resource monitoring and notification, application developers still have to make decisions on when and how to adapt to different network conditions.

Although sufficient for some scenarios, end-point approaches are rather limited in three ways. First using only server and client nodes in adaptation may not be flexible enough to cope with changes in intermediate links. For example, if the bandwidth of a communication path drops as a result of an increased error rate at an intermediate link (consider a wireless link getting affected by bad weather conditions), a typical end-point solution for this might introduce compression/decompression operations at server and client nodes respectively, which may end up not increasing the achieved bandwidth by much. A better solution would be to deploy error detection and recovery functions at both ends of that link to quickly respond to packet transmission errors, which can further take advantage of *local knowledge* of the link characteristics. Second, tight coupling between client and server nodes may considerably complicate the logic of both servers and client applications. And lastly, end-point approaches usually need to make some assumptions about capacity of servers or client nodes, which may not hold on resource-constrained sites/devices. We will revisit this point later in Chapter 9.

Proxy-Based Approaches

In a *proxy-based approach*, shared proxy nodes, instead of server nodes, are used to handle different network conditions.

The cluster-based proxies in BARWAN/Daedalus [18], TACC [19], and MultiSpace [22] are examples of systems where application-transparent adaptation happens in intermediate proxy nodes in the network. Active Services [2] permits a client application to explicitly start computation agents on its behalf on a gateway node for transforming the data it receives from an end service.

Similarly, Ninja [21] proposes the use of cluster based proxies, which are usually placed before the last hop, to do aggressive computation such as content distilling and transcoding on the fly to cope with variations on the client side (i.e. network, hardware and software used by clients).

A more interesting aspect of Ninja is the Ninja Automatic Path Creation (APC) service, which is also used in the Universal Inbox infrastructure [47]. APC is used to create paths between various end devices and services. A Ninja path, which runs on a proxy site and provides applications with data of a required format, consists of a sequence of components. Although Ninja APC can automatically create communication paths to handle (static) variations on the client sites, the paths created in Ninja are somewhat limited. At a high level, APC is a function-oriented method, which ignores network link properties, network resource constraints, and dynamic resource availability, therefore the application performance improvements achieved using such paths are also very limited. No dynamic reconfiguration of paths is supported in Ninja.

Compared with end-point approaches, the proxy-based approaches offer their own advantages and disadvantages. First, limiting adaptation to occur only at proxy sites relieves servers from this task, thus simplifying logic on the server side. Second, as mentioned in [18], managing a large proxy site to do adaptation, which can be shared by a large number of servers, is more economically efficient than managing each of these servers individually. Last, because of the resource sharing at the proxy sites, such approaches can work with server site or end devices that have insufficient capacity because they can take advantage of shared resources at the proxy sites. On the other side, disadvantages of the proxy-based approaches are also obvious. First, similar to the end-point approaches, the proxy-based approaches cannot handle local

changes very well. Second, the limitation that adaptation only occurs at proxy sites may result in considerable network resource waste before the last hop, especially for long communication paths (recall the example discussed in Section 1.3).

Path-based Approaches

Differing from end-point and proxy-based approaches, recently several frameworks have proposed the injecting of functionality along the whole communication path to address the problems caused by different network conditions. In this more general view, any node along a communication path can participate in adaptation.

Active Names [53] is such a framework for deploying a sequence of programs along a communication path by intercepting the name resolving procedure. The active name framework has a hierarchically organized name space; as a name request from a client is being resolved, the name services construct a chain of programs for transporting data back to the client application. Though listed here as a path-based adaptation framework, the focus of Active Names is mostly on general mechanisms for injecting general functionality into communication paths, it does not provide much support for enabling applications to adapt to different network conditions.

In the Conductor project [59], multiple application-transparent components (called adaptors) can be automatically deployed along the communication path between a application and a service. The transparency (without application input) implies that such systems need to rely upon self-describing properties of data streams and the data format required by the client needs to be exactly the same as what is provided at the server side. The first assumption may or may not hold given increasingly proprietary content. The second assumption, though it considerably simplifies functions required

in the communication path, severely limits the applicability of such systems (consider a mobile device that requests WML pages from a web site that provides only HTML data). Conductor contains a planning scheme for placing adaptors to augment an application's data stream to address unfavorable network conditions. While two schemes are discussed in [48], one based upon selection from a reusable plan set and the other based on exhaustive constraint space-based search, to the best of our knowledge these schemes have not yet been implemented or evaluated with real applications.

Recent work in the Scout project [43] has extended the path notion from a data flow within a single node system to one that traverses across networks. The approach it uses for building such paths is still a template-based algorithm, which takes into consideration the resource requirements (for delivering media objects), user preferences, node capabilities, and programmer-provided path rules. Though such an approach can be used to improve performance of applications, it requires a priori construction of path templates and storing them into a central database, simply choosing an appropriate template and instantiating it based on other programmer-provided rules that decide whether or not a component can be created on a resource.

Kiciman and Fox [36] have proposed a general path infrastructure framework for composing mediators distributed across a network of machines. This infrastructure is built upon Ninja's APC service and suffers from the same limitations. Furthermore, this approach separates out logical path creation (choice of components) from the mapping of components to physical resources. Although this separation considerably simplifies the problem of creating paths across multiple network resources, it can result in poor performance for generated paths since these two stages are usually tightly interrelated.

Compared with end-point and proxy-based approaches, the primary benefit of using path-based approaches stems from the flexibility that all segments along a communication path can respond to dynamic changes in the network. Consequently, such local adaptation can result in better agility. Besides, similar to the proxy-based approach, path-based approaches benefit from resource sharing, which is much more flexible because shared resource pools are set up across the whole network instead of being limited to proxy sites only. On the other hand, building a path-based solution is much more challenging because a long communication path may involve multiple different network domains. Consequently, centralized schemes usually do not perform well, thus support for distributed path construction and maintenance is required.

Despite the existence of these frameworks mentioned above, current work on path-based approaches is very limited and many challenging problems still remain open. In particular, the following questions need to be resolved before such an approach can be used to realize our vision of network-aware data communication as described in Chapter 1.

- How should one model the impact on data communication of components and network resources along a communication path so that valid structures can be identified mechanically?
- How does one construct *optimal* communication paths according to application performance requirements and underlying network conditions?
- How does one provide continual and efficient adaptation to dynamic changes in the network by dynamically modifying communication paths?
- How does one enable each segment of a communication path to be indepen-

dently and concurrently responsive?

- How does one efficiently manage network resources?
- What fundamental advantages does the path-based approach bring as compared to other alternatives?

Building an infrastructure that provides network-aware communication paths using the path-based approach has to answer all these questions. This motivates the work described in this dissertation.

2.2.4 Summary

As the Internet exhibits increasingly complicated behaviors and the diversity of Internet applications grows, network awareness in data communication becomes indispensable for delivering satisfactory performance.

Though general mechanisms that introduce more functionality into the network layer can enhance performance of data communication to some extent, the lack of information about applications and data in transmission severely limits the network awareness achieved with such approaches. General adaptation frameworks built on the application layer can provide better performance. While network awareness can be realized in different ways, i.e. end-point, proxy-based, or path-based approaches, the path-based approach is the most promising way for realizing our vision of network-aware data communication. However, many challenging problems need to be addressed before this vision can become reality. This dissertation presents our solutions to these problems.

Chapter 3

Architecture

In this chapter, we present an overview of our path-based framework. We first introduce our logical view of the network, then describe the concepts of components and augmented communication paths. After that, we revisit the set of open problems that have to be addressed before network-aware data communication can be realized using a path-based infrastructure, and briefly discuss our solutions for these problems.

3.1 Logical View of the Network

Our framework takes a general view that the network consists of *applications*, *services*, and *communication paths* connecting the two. The notion of the communication path is extended from one traditionally limited to data transmission between end points to include application-specific functionality dynamically injected by end services, applications, or the underlying infrastructure. Such functionality takes the form of components, which are self-contained pieces of code that can perform a particu-

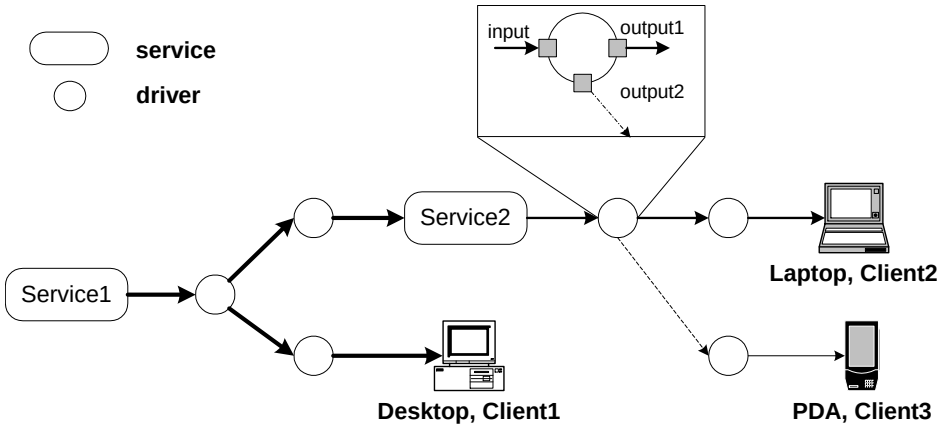


Figure 3.1: Logical view of a network showing data paths constructed from components.

lar activity, e.g., protocol conversion or data transcoding. Components are connected with each other at run time and operate on data streams to provide network awareness in data communication by matching application requirements with physical characteristics of the underlying network and properties of end devices (see Figure 3.1).

Our framework is realized in Execution Environments (EE), an instance of which runs on all infrastructure-enabled nodes. Augmented paths are deployed to these nodes. The execution environment provides interfaces for applications to create and manage paths, and an environment for component execution, basically serving as the underlying "operating system" of our infrastructure.

3.2 Components

Components serve as the basic building block for constructing adaptation-capable, augmented communication paths. A component is a standalone mobile code module that performs a single operation on the data stream. We sometimes refer to compo-

nents as *drivers*, using the terms interchangeably. Augmented paths are constructed by dynamically composing components. To enable efficient composition and dynamic low-overhead reconfiguration of augmented paths, drivers are required to adhere to a common interface as shown in Figure 3.2 and provide the following properties:

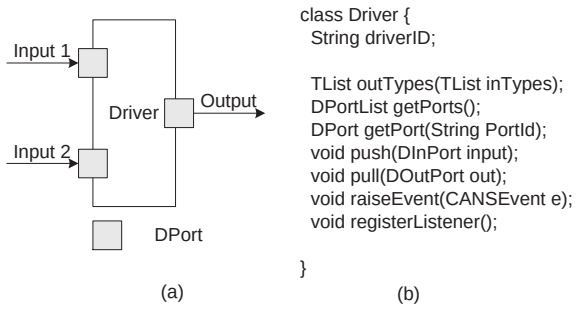


Figure 3.2: Driver functionality (a) and interface (b).

1. Drivers consume and produce data using a standard *data port* interface, called a `DPort`. `DPort`s are associated with type information (details deferred to Chapter 4) and distinguished based on whether they are being used for input or output. Information about data ports and their types can be queried at run time.
2. Drivers are *passive*, moving data from input ports to output ports in a purely demand-driven fashion. Driver activity is triggered only when one of its output ports is checked for data, or one of its input ports receives data.
3. Drivers consume and produce data at the granularity of an integral number of application-specific units, called *semantic segments*. These segments are naturally defined based on the application, e.g., an HTML page or an MPEG frame.
4. Drivers contain only *soft state*, which can be reconstructed simply by restarting the driver. Stated differently, given a semantically equivalent sequence of

input segments, a soft-state driver always produces a semantically equivalent sequence of output segments, even after the internal state of the driver gets reset.

The first two properties enable dynamic composition and efficient transfer of data segments between multiple drivers that are mapped to the same physical host (e.g., via shared memory). Moreover, they permit driver execution to be orchestrated for optimal performance. For example, a single thread can be employed to execute, in turn, multiple driver operations on a single data segment. The overhead of invocation between different drivers is basically a few function calls, as if driver operations were statically combined into a single procedure call. The only extra overhead compared to using a statically linked module is the overhead of using virtual functions. Finally, this choice greatly simplifies and enhances the efficiency of resource management among multiple paths, enabling control over resource consumption of individual paths within an execution environment.

The semantic segments and soft-state properties enable low-overhead dynamic adaptation, either within a single driver or across communication path segments while preserving application semantics, a topic discussed in more detail in Chapter 6. The last thing deserving mention here about the driver interface (see Figure 3.2) is the methods that permit a driver to raise and listen to events, facilitating its participation in distributed adaptation activities.

3.3 Augmented Communication Paths

Unlike conventional communication paths, an augmented path in our framework contains functionality to process data in an application-specific fashion. Introducing such functionality into communication paths can bring application two major benefits. First, they can be used to match application requirements with the underlying network conditions. For example, compression functionality can be used for addressing the problem of low bandwidth in a network link; encryption functionality can be applied to address problems caused by network links that do not provide sufficient guarantees on data privacy and integrity. Second, by allowing computation in communication paths, functionality of an application can be extended with what exists in the network. For example, for a small device that can only display WML pages but needs to access an Internet service where only HTML format is supported, the augmented communication path can handle the conversion from HTML to WML by orchestrating functionality in the network, so that the browser running on the devices can display the contents appropriately.

To construct network-aware communication paths, we need a way to orchestrate various kinds of functionality together. Instead of using a monolithic implementation, our approach adopts a much more extensible approach where communication paths are constructed by dynamically composing different components. This composition approach allows application development to be completely separated from component authoring, which itself is decoupled from other components as well. More importantly, it provides a foundation that allows the construction and dynamic reconfiguration of such augmented paths to be managed by the underlying infrastructure

without requiring involvements from applications.

Network awareness in data communication is realized by these augmented paths and the underlying infrastructure: application specific functionality is included in the augmented communication paths; the underlying infrastructure is responsible for creating and controlling them to continually adapt to changes in the network, in accordance with application performance requirements.

To construct such augmented paths, only high level information is required, which includes services properties, application requirements, and characteristics of the underlying platform. The components that constitute a communication path, the interconnections amongst them, and their internal configuration parameters can all be modified by the infrastructure at run time to cope with different network conditions, when application requirements can not be met by the current configuration given the resources available.

3.4 Open Problems in Previous Path-Based Adaptation Infrastructures

As mentioned in Chapter 2, building a path-based infrastructure that provides network-aware data communication needs to address the following problems: modeling of application specific functionality and network resources, path creation and reconfiguration strategies, and network resource management. Furthermore, since the infrastructure is targeted at wide area networks, distributed solutions that do not require global knowledge or centralized entities are required. Here we revisit the questions identified in Section 2.2.3, briefly sketching our solutions for them.

3.4.1 Type-based Modeling

Question 1: How should one model the impact on data communication of components and network resources along a communication path so that valid structures can be identified mechanically?

Carefully modeling component behaviors and the effects of different network resources along a communication path is necessary for mechanically identifying valid structures of augmented paths, which provides a foundation for automatic creation of network-aware communication paths.

Our framework uses a high-level integrated *type-based specification of components and network resources*. Components are modeled as a mapping between different set of types. Composibility between different components is determined by the type compatibility of the components. The aggregate effect of component composition is depicted using a notion of *stream type*, which eliminates the need for complete knowledge of the entire communication path when only parts of the path need to be modified. The effects or constraints introduced by network resource characteristics are modeled using the notion of *augmented type*. Application specific composition constraints are expressed using a *type ranking* scheme.

Differing from conventional static type models, values of type instances in our framework are calculated at run time, i.e. a component defines its own function for calculating the outgoing type values given incoming type instances. Type values automatically flow downstream when components are connected together. This feature is important for enabling late binding of components to paths, essential for flexibility of dynamic composition. Chapter 4 describes the type model in more detail.

3.4.2 Automatic Path Creation Strategies

Question 2: How does one construct *optimal* communication paths according to application performance requirements and underlying network conditions?

To construct network-aware communication paths, the most important question is that given the resource availability and application requirements (on data format and performance), how does one select the path that can provide the best performance.

This dissertation describes *automatic path creation strategies* suitable for this purpose. In addition to providing the required data format, generated paths also provide applications with optimized performance for the underlying network conditions. Our path creation strategies are very flexible: they can be used with applications that have different type of performance requirements (i.e. a maximum/minimal value or an acceptable value range of a performance metric); they can be used for creating a whole communication path or replacing a small portion of an existing path. Furthermore, our strategies have distributed solutions for calculating communication paths across different network domains, i.e. a path can be incrementally constructed from one network domain to another without requiring a central entity or complete knowledge of the whole network. The last two properties are very important for any path-based infrastructure, especially for those that need to be deployed in a wide area network. Chapter 5 describes our path creation strategies in detail.

3.4.3 Support for Path Reconfiguration

Question 3: How does one provide continual and efficient adaptation to dynamic changes in the network by dynamically modifying communication paths?

As underlying network conditions change dynamically, a network-aware communication path needs to adjust its configuration accordingly. We refer to the procedure of adjusting the current path configuration as *path reconfiguration*. An ideal solution for reconfiguring a network-aware communication path should 1) avoid introducing a long interruption period into the data transmission, 2) provide semantic continuity guarantees. The problem of semantic continuity stems from the fact that at reconfiguration time there may exist data in the network or as internal state inside the components along the path being reconfigured.

Our framework contains system support for *low-overhead dynamic path reconfiguration*, which has two major parts. The first part is a set of simple rules that components are required to conform to. The second part is a reconfiguration protocol that is used to modify communication paths while maintaining semantic continuity of data transmission by exploiting component properties derived from those rules.

Path reconfiguration in our framework is completely controlled by the infrastructure. Moreover, path reconfiguration can be conducted on an entire communication path as well as on multiple disjointed segments independently and concurrently. The latter is referred to as *local reconfiguration*. Using local reconfiguration can result in better responsiveness to local changes in the network, besides it also greatly reduces the need for coordination across different network domains, which makes the infrastructure suitable for highly decentralized environments.

Combining our path creation strategies and reconfiguration support, fine-tuned and desirable adaptation behaviors can be provided to regular applications without requiring onerous effort from application developers. Detailed description of path reconfiguration appears in Chapter 6.

3.4.4 Resource Management

Question 4: How does one efficiently manage network resources?

Resource management strategies for a path-based infrastructure have to provide solutions for two questions. The first question is how to allocate resource capacity among multiple active paths. Since an augmented communication path usually involves multiple shared network resources, the goal is to support as many path as possible and provide individual paths with the best possible performance. Since our framework is targeted at a wide area network, we design a distributed scheme where individual network resources can make their own decisions without requiring expensive coordination among different network domains. The scheme can improve both individual path performance and resource utilization of the whole network.

The second question is how to set up shared resource pools across the network for a path-based infrastructure. The goal is that given a fixed amount of computation resource, we need to optimize the overall performance of the whole network. The scheme used in our framework takes into account the existing organization of Internet-like networks, and provides a model and algorithms for distributing computation resources hierarchically across the network (i.e. moving computation resources from low-level network domains to high-level ones). By setting up shared resource pools at high-level nodes in the network graph, the overall performance of the whole network can be improved because overloaded portions can take advantage of spare resources from others. Our scheme is able to set up a maximal resource pool at high-level network domains without compromising the performance of low network domains from which the computation resources are moved out. Detailed description of

our resource management strategies appears in Chapter 7.

3.5 Assumptions in Our Framework

Two assumptions exist in our framework.

First, our framework does not address trust and security issues. Note that although the distributed version of path creation and reconfiguration strategies in our framework considerably reduces interdependency between different network domains, such mechanisms are still needed for sharing information (types) and code among them. Second, we assume resource monitoring functionality is provided by entities external to our framework.

These assumptions are relatively independent from the network awareness focus of this dissertation. Furthermore, for both of these issues, there is a considerably large amount of literature available that points to how appropriate solutions can be constructed. We defer a detailed discussion about these solution to Section 10.3.

3.6 Summary

In this chapter, we have presented the overall architecture and key concepts of our framework, which provides applications with network-aware communication paths. Network awareness is realized using communication paths that are augmented with application specific functionality and an infrastructure to manage these augmented paths. The functionality built into such augmented paths is used to match application requirements to underlying network conditions. Moreover, such paths can dynami-

cally adapt to dynamic changes in the network. Since path creation and adaptation are completely handled by the underlying infrastructure, regular (legacy) applications can easily be augmented with adaptation capability without onerous effort from application developers.

To realize this vision, our framework relies on four key schemes: type-based modeling of network resources and components, automatic path creation strategies, system support for path reconfiguration, and distributed strategies for managing network resources. Our schemes can be used in wide area networks because both distributed and local operations are provided.

Chapter 4

Type Model

In this chapter, we present the type model that serves as the foundation of our automatic path creation strategies. Since network-aware paths in our framework are realized as compositions of different components, in order to identify valid composition patterns mechanically (enumeration is not feasible for most cases), we need a model to describe the effects of functionality built into those components and network resource characteristics along communication paths. Our approach for this is a high-level type model that is used for abstracting component behaviors, network resource characteristics, and expressing various composition constraints.

Generally speaking, a valid composition pattern may have to satisfy three properties. First, it should provide the exact data required by the application. The requirements may include not only data format, but also other properties such as privacy guarantees etc. Second, all connections between adjacent components should be valid, i.e. data produced by the upstream component can be processed by the downstream one. Third, the composition order among these components should not

violate any specified rules. For example, encryption operations are usually required to appear after compression operations but not vice versa.[1] Furthermore, since network awareness in our framework is realized throughout the whole communication path, of which each segment can adapt to changes independently and concurrently, schemes that can work with path segments without requiring a complete knowledge of the whole communication path are needed.

In our framework, all these requirements are expressed and enforced using a unified type model. The basic idea is the notion that all data flowing along a communication path is *typed*, and that values of type instances are affected by components along the data path as well as network resources making up the route. We model the functionality of a component as a mapping from input types to output types. Composibility between different components is modeled as a type compatibility problem between those components. The effect of network resources on the communication path is captured with a notion of *augmented types*. The aggregate effect of components on the path is captured using the notion of *stream types*, therefore knowing the incoming and outgoing stream type values is sufficient for understanding the functionality within a path segment, eliminating the need for complete knowledge of the whole path. Type ranks, which constrain possible structures of stream types, are used to express specific constraints on composition orders among components.

4.1 Modeling Component Functionality

Types associated with components include two concepts: *data types* and *stream types*.

[1] Encrypted data usually cannot be compressed effectively.

Data type is the basic unit of type information, represented by an object that in addition to a unique name can contain arbitrary attributes and a method for checking type compatibility. Our framework assumes that, in most application domains, it is possible to define a *closed*, semantically unambiguous set of types, e.g., MIME types to represent common media objects.

Traditional type hierarchies can still be used to organize data types. However, realizing type instances as objects with the compatibility method give us the utmost flexibility in defining type compatibility relationships that cannot easily be expressed just by matching type names. For instance, it is possible to define a customized `MPEG` type, which contains a frame size attribute such that it is compatible with any `MPEG` types with smaller frame size (shown in Figure 4.1), naturally capturing the behavior that a lower resolution `MPEG` stream can be played on a platform capable of displaying a higher resolution stream.

Stream types capture the aggregate effect of multiple drivers operating upon a data stream. Stream types are constructed at run time, and represented as a *stack* of data types. For example, after an `MPEG` type passes through an encryption driver (Figure 4.2), the stream type of its output port is a stack in which the type `Encryption` is placed on top of the type `MPEG`.

The primary reason for using stream types is for eliminating the requirement for complete knowledge of the whole path when small portions of the path need to be adjusted independently. By using stream types, any segment of a path only needs to consult its incoming and outgoing stream type instances.

This point is highlighted in Figure 4.2 in which an `MPEG` type passes through an

```
public class MPEGType {
    public boolean isCompatible(DataType dt) {
      ...
    }
    ...
    int height, width;
}

public class MyMPEGType extends MPEGType {
    public boolean isCompatible(DataType dt) {
        if((dt instanceof MPEGType)
            &&(((MPEGType)dt).width<=width)
            &&(((MPEGType)dt).height<=height))
           return true;
        else return false;
    }
    ...
}
```

Figure 4.1: An Example of the Type Compatibility Method

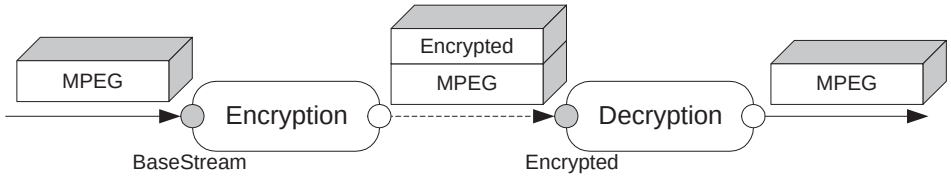


Figure 4.2: An Example of Stream Types

`Encryption` driver and a `Decryption` driver. If components were just modeled as consuming data of a particular type and producing data of another, it would be difficult to express the behavior of the `Encryption` and `Decryption` drivers in a way that permits their use with generic types *without* losing information about the original type at the output of the `Decryption` driver. Specifically, without stream types, the *Encryption* driver will set its output as being of the `Encrypted` type, and the output of the `Decryption` driver ends up being of the `BaseStream` type (unless the en-

tire communication path is examined). This will cause a type compatibility problem at some downstream point because the client requires a more specific type (`MPEG`) than the incoming type (`BaseStream`). In contrast, the stream type representation permits local decision making, which is important for run-time adaptation via dynamic component composition, especially for the cases where long communication paths are used.

Operations allowed on stream types include *push, pop, peek,* and *clone,* which have the standard meanings. From the type point of view, each CANS component with m input ports and n output ports defines a function that maps its input stream types into output stream types: $f(\mathrm{DT}_{\mathrm{in}_1}, \mathrm{DT}_{\mathrm{in}_2}, ..., \mathrm{DT}_{\mathrm{in}_m}) \rightarrow (\mathrm{ST}_{\mathrm{out}_1}, \mathrm{ST}_{\mathrm{out}_2}, ..., \mathrm{ST}_{\mathrm{out}_n})$ where $\mathrm{DT}_{\mathrm{in_i}}$ is the required *data type* set for the ith input port, and $\mathrm{ST}_{\mathrm{out_j}}$ is the resulting *stream type* produced on the jth output port. The type compatibility between an input and an output port is determined by checking the top of the output port's stream type against the required data type of the input port. Stream type information flows downstream automatically when two ports get connected at run time.

4.2 Modeling Network Resource Characteristics

In addition to the effect of components along a communication path, network resource characteristics can also have impacts on paths. This consequently introduces additional constraints affecting both which components must be present along a communication path and how these should be composed. For example, the risk of packet interception on a shared wireless link necessitates the presence of a pair of encryption and decryption drivers for preserving privacy of data transmission. Since these

drivers are not required if one just examines the type properties of the data source and that required by the client application, it is clear that one needs to take into account network resource characteristics into the component selection process.

In our approach, such constraints are also described by our type model as a type compatibility problem. Modeling such constraints along with composability of components in the same type model is very important for the automatic path creation strategies described in the next chapter, which strive to find the optimal communication path for underlying network conditions. In the following discussion, we restrict our attention to network links, but the same principle can be applied to other network resources. The basic idea of our approach is to represent requirements for specific components because of link characteristics implicitly by modeling how links affect the types of data that go across them.

To capture the effect of link properties on data types, we introduce the notion of an *augmented type*: each data type is extended with a set of link properties such as security (used here to denote transmission privacy), reliability, and timeliness, etc. These properties can take values from a fixed set (boolean values for most properties). Network links are modeled in terms of the same property set and have the effect of modifying, in a type-specific fashion, values of the corresponding properties associated with different data types. To give an example, consider transmission of `MPEG` data over an insecure link. Our type framework captures this as follows: the data type produced at the source is represented by `MPEG` (`secure=true`), the network link is represented by the property `secure=false`, and the effect of the link property `secure` on the `MPEG` data type by the rule that the augmented type `MPEG` (`secure=true`) is modified to `MPEG` (`secure=false`) upon crossing a link with

```
Typedef LinkProperties AugmentedPart;

public class DataType {
 protected AugmentedPart ap;
 ...
}

public class StreamType {
     public void passLink(LinkProperties linkProp) {
        for(Iterator i=typeStack.iterator(); i.hasNext(); ) {
            DataType dt=(DataType)i.next();
            dt.passLink(lp);
        }
     }...
     private Stack typeStack;
}

public class MPEGType extends DataType{
    public AugmentedPart passLink(LinkProperties linkProp) {
        ap.security&=linkProp.security;
        ...
     }...
}

public class EncryptedType extends DataType{
    public AugmentedPart passLink(LinkProperties linkProp) {
        //isolation of the security property
        linkProp.security=true;
        ap.security&=linkProp.security;
        ...
     }...
}
```

Figure 4.3: Code fragments showing use of Augmented Types

the property `secure=false` (shown as the `MPEGType` class in Figure 4.3).

This base scheme is extended to stream types by introducing the notion of *isolation*. Stated informally, specific data types have the capability to isolate others below them in the stream's type stack from having their properties being affected by a link. For example, an `Encrypted` type can isolate the `secure` property of types that it

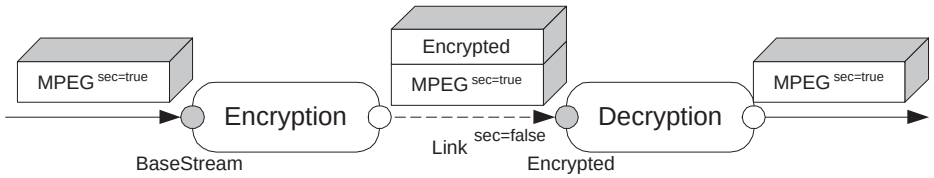


Figure 4.4: An Example of Augmented Types and the Isolation Effect

"wraps", i.e., this type of encrypted data still remains secure after crossing insecure links, irrespective of what specific type(s) the data corresponds to. The reason that isolation only works for the types below the "wrapping" type in a type stack is because the effects caused by components afterwards can not be covered. For example, an encryption driver cannot provide privacy guarantee for any data that is appended after the encryption operation.

The `EncryptedType` class in Figure 4.3 shows an example of how the isolation notion works. When a stream type passes through a network link, the method `StreamType.passLink` will be invoked, which in turn calls the `passLink` method upon each type in the type stack (from the top to the bottom). The isolation effect occurs when the `passLink` method of the `EncryptedType` is invoked, forcing the `security` property of the link to be set to `true`. This means that for all type instances below the `EncryptedType`, their security property will not be affected by the link, i.e. will be the same value as at the source.[2] Figure 4.4 shows how these concepts work together by depicting the case in which an `MPEG` type passes through an unsecured link using an `Encryption` driver.

In addition to security properties, this scheme can also be applied to other network

[2]If the security property of the source is false, it will remain as false even with an encryption, which reflects the fact that the privacy guarantee has already been compromised before that link.

characteristics, such as reliability, timeliness etc. A case study described later in this chapter provides more details about how to use this scheme.

4.2.1 Modeling Constraints on Composition Order

To express constraints on the order of composition, we use the notion of *type ranks*: if type t_1 and t_2 satisfy $\mathrm{rank}(t_1) > \mathrm{rank}(t_2)$, then t_1, t_2 cannot appear on the same type stack with t_1 appearing below t_2. This simple scheme can be used to express various constraints on how components can be composed together.

For instance, assigning the encryption type a higher rank ensures that for any communication path requiring both encryption and compression, encryption will always happen after compression. Similarly, the ranking scheme can be used to describe the constraint that a relatively stronger compression can happen after a relatively weaker one but not vice versa.

To simplify the use of types, our infrastructure predefines a set of commonly used data types for operations such as encryption, compression, image transcoding etc. These types are organized into a linear rank lattice. When a new type is added in the lattice, its constraints on type ranks will also be automatically checked by the system. Constraints on composition order, of course, can also be expressed using some rule-based mechanisms; however our scheme is simple to use with our type model and quite expressive in describing various composition constraints. By using the notion of type ranks, valid composition patterns can be identified by only checking type compatibility between adjacent components and the type stacks that appear along the communication path.

4.3 Case Study: A Streaming Media Application

To demonstrate how such type-based modeling can be used to identify **valid** augmented communication paths, we describe the following scenario. Consider a mobile user, using a laptop with both wired and wireless connections, who downloads a media stream from an Internet-based server. This user starts off at his office desk but then has to leave in the middle to go elsewhere in the building. Let us assume that the user wishes to continue viewing the stream using the laptop's wireless connection, while retaining the same privacy guarantees (freedom from eavesdroppers) he might have had on a wired connection even if, as we assume here, the wireless link provides inadequate security guarantees.

An ideal network-aware communication path would provide the user with a stream of high quality when he is using a wired connection, and the quality gracefully degrades depending on his distance from the wireless access point. Additionally, the path would isolate the user from the switch between wired and wireless connectivity, transparently providing the required privacy guarantees.

The type-driven view in our framework can identify valid communication paths that enable this scenario by augmenting the path between the user and media server with the following six components: `reconnector(src)`, `reconnector(dest)`, `padder`, `splitter`, `encryption`, and `decryption`. The `reconnector(src)` and `reconnector(dest)` components cooperate to buffer and retransmit frames of the stream, ensuring that the client application always receives semantically valid frames despite any connection disruptions. The `padder` component "fills in" legal media frames whenever its input stream stops, and helps isolate the media player ap-

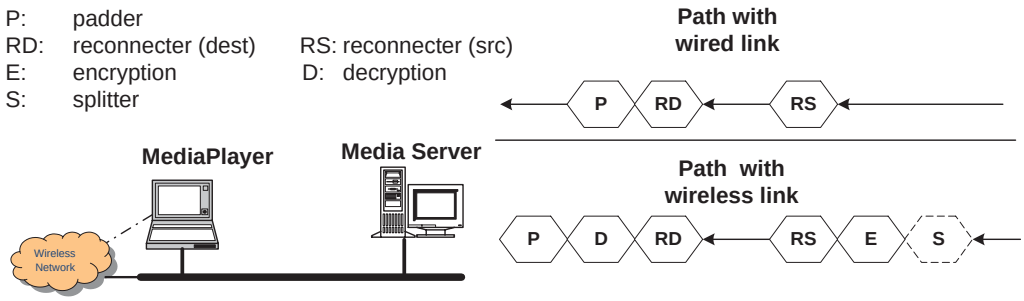


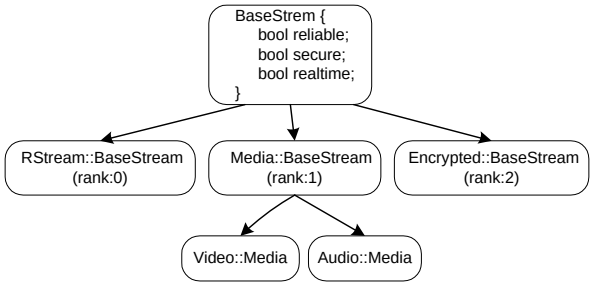
Figure 4.5: Valid communication paths for a Mobile User to Access a Media Server

plication from any reconnection delays. The `splitter` component can split the incoming media stream into its video and audio portions, enabling adaptation in low-bandwidth situations. Finally, the `encryption` and `decryption` components co-operate to preserve privacy of stream data by encrypting it before the wireless link and decrypting it before delivering to the application.

4.3.1 Type-based Modeling

To identify valid communication paths the example application described above, the specification of components need to include the following four pieces of information: data type definitions (including rules governing how data types are modified by links), network links modeled in terms of a set of link properties, and component properties described in terms of input and output types.

Figure 4.6(a) shows the data type definitions. `BaseStream` is the basic stream type with three boolean link properties: `reliable`, `secure`, and `realtime`. The types `RStream`, `Media`, and `Encrypted` extend the `BaseStream` type, representing reliable, media, and encrypted streams respectively. `Video` and `Audio`



(a)

	properties		
	`secure`	`reliable`	`realtime`
wired	T	F	F
wireless	F	F	F

(b)

	`secure`		`reliable`		`realtime`	
	T	F	T	F	T	F
`Media`	—	F	—	F	—	F
`RStream`	—	F	T*	T*	—	F
`Encrypted`	T*	T*	—	F	—	F

—: no change *: Isolation Effect

(c)

components	Input & output type	components	Input & output type
Media player (sink)	media:{ realtime = T; reliable = T; secure = T; }	encryption	Encrypted * *
source	video:{ realtime = T; reliable = T; secure = T; }	decryption	* Encrypted *
splitter	audio video	reconnecter (src)	RStream * *
padder	media:{ realtime = T; } media:{ realtime = *; }	reconnecter (dest)	* RStream *

(d)

Figure 4.6: Types in the streaming media example: (a) data type definitions; (b) link properties; (c) effect of link properties on augmented types; and (d) input and output types of components.

are two subtypes of the `Media` type. The `RStream` type is given a lower rank as compared to the other types to capture the composition constraint involving the `encryption`/`decryption` and `reconnecter` drivers.

Figure 4.6(b) shows properties of the wired and wireless links. The wired link is modeled with `reliable` and `realtime` properties set to `false` to capture the fact that it can get disconnected during the access. Similarly, the wireless link has the `secure` property set to `false` to denote its limited support for transmission privacy.

Figure 4.6(c) shows how these link properties affect different types. For example, the `security` property of the media type will be changed to `False` after it passes through a link whose `security` property value is `False`, but the value remains the same if the security property value of the link is `True`. As discussed in Section 4.2, some of the types have the effect of isolating certain link properties from those below them in the type stack. In this example, the `Encrypted` type isolates the security property, and the `RStream` type isolates the reliability property.

Figure 4.6(d) lists the input/output types of the six components, along with the types produced by the source and that required by the sink. To consider some examples, the sink specification says that the client application requires a reliable, real time, and secured `Media` type. The `padder`, which fills in legal frames whenever it does not receive input in a timely fashion, is represented as a component that transforms the input type `Media` with an arbitrary value for the `realtime` property, into the output type `Media` with `realtime=true`. Similarly, the `encryption` component is modeled as an entity that converts an arbitrary stream type at its input into a new stream type consisting of the `Encrypted` type wrapping whatever was originally present. The `decryption` component performs the reverse operation, stripping away the `Encrypted` type out of the stream type.

4.3.2 Valid Paths

The primary advantage of modeling component properties, network resource characteristics, and composition constraints in a unified type model is that all valid communication paths associated with a given set of network conditions correspond simply to type-compatible component sequences that transform the source data type into that required by the sink. The important point here is that these valid sequences can be inferred fully automatically. In this example, the two network conditions of interest are whether the user connects to the server using a wired link or a wireless link.

With the wired link, the above type specifications yield the following two valid component sequences: `reconnecter(src)`—`reconnecter(dest)`—`padder`, and `splitter`—`reconnecter(src)`—`reconnecter(dest)`—`padder`. Informally, the former might be used when link capacities are sufficient for transmission of the original video+audio stream to the client, while the latter is required when this is not the case.

With the wireless link, we also have two valid sequences: `encryption` — `reconnecter(src)`—`reconnecter(dest)`—`decryption`—`padder`, and `splitter`—`encryption`—`reconnecter(src)`—`reconnecter(dest)`—`decryption`—`padder`. Notice that the `encryption` and `decryption` components are required to preserve the secure property of a stream transmitted across the wireless link (see Figure 4.6(c)). Note also that an alternate type-compatible sequence `reconnecter(src)`—`encryption`—`decryption`—`reconnecter(dest)` —`padder` is disallowed because of the ranks associated with the `RStream` and `Encrypted` types.

4.4 Summary

In this chapter, we have presented our type-based model of component functionality, network resource characteristics and composition constraints. Components are modeled as entities that transform data from input types to output types. Aggregate effect of component composition is represented using the notion of stream types, which eliminates the need for complete knowledge of the whole communication path when modifying only a subset of the path. When passing through a network resource, the augmented part of data types may be changed depending on the characteristics of the resource, consequently introducing additional constraints for meeting application requirements. Constraints on composition order are expressed using type ranks. Our type model and these concepts serve as the foundation of our path creation strategies, described in the next chapter, which provides a mechanism for validating communication paths mechanically.

Chapter 5

Automatic Path Creation Strategies

In this chapter, we present our path creation strategies for automatically generating augmented paths to meet application requirements, while providing optimized performance for the underlying network conditions.

In general, creation of an augmented communication path consists of two steps: *route selection* where a graph of nodes and links is selected for deploying the path, and *component selection and mapping* where appropriate components are selected and mapped to the chosen route. Route selection is typically driven by external factors (such as connectivity considerations of wireless hops, ISP-level agreements, etc.) and so we focus only on the component selection and mapping problem here.

The component selection and mapping process takes as input the application communication requirements and a chosen route between client and server nodes, and produces a sequence of drivers and their mapping to the route that can provide the application with optimized performance. Given that our goal is to provide network awareness to applications in a *wide area network*, the strategies described in this chap-

ter have the property that they can also work with small segments of a communication path and can be employed incrementally across different network domains in a distributed fashion.

To optimize path performance, we first need ways of characterizing the impact of a particular component on the resource utilization along a path, as well as for associating a performance metric with the overall path.

5.1 Performance Characteristics of Network Resources and Components

In this section, we describe how we model performance characteristics of network resources and components along an augmented path.

5.1.1 Network Resources

Performance characteristics of a network resource are modeled in terms of its capacity: computation capacity (i.e. how many operations per time unit) for a network node, bandwidth and latency for a network link. An individual path is allocated a certain share of the resource, in accordance with the resource management algorithms described in Chapter 7.

5.1.2 Component resource utilization model

To characterize the resource utilization and performance of a path, we need to capture the behavior of each component without requiring an explicit enumeration of all possible situations in which the component can be mapped. To facilitate this,

each component c is modeled in terms of its *computation load factor* (**$\mathrm{load}(c)$**), the average per-input byte cost of running the component, and its *bandwidth impact factor* (**$\mathrm{bwf}(c)$**), the average ratio between input and output data volume. For example, a compression component that reduces stream bandwidth by a factor of two has a $\mathrm{bwf} = 0.5$. The value of load and bwf can be obtained via profiling with typical data input.

This component model assumes that the underlying behavior of the component, with respect to computation time and output data size, varies linearly with input data size. Knowing the rate at which input data packets arrive at the component permits one to estimate the CPU requirements as well as network bandwidth requirements on the downstream link.

We should note that our algorithms themselves do not rely on this specific linear model of $\mathrm{load}(c)$ and $\mathrm{bwf}(c)$: the computation load or compression ratio can be any arbitrary function as long as this information can be provided to our planning algorithms by components themselves or by detailed profiling. The primary reason that we choose computation load to be a linear function of the input data size and compression ratio to be a constant value is because of profiling experiments we have conducted with typical components. Appendix A lists the raw profiling data and shows that it is represented well by this model .

This simple model can be extended to allow components to have multiple configurations. Further, for each configuration, the values of computation load and compression ratio can be parameterized by the actual stream type of incoming data. For example, when an image resizing component is placed after an image filter, its load and bwf factors are determined by the image quality attributes contained in the type

object generated by the filter. Such values can be obtained by an approach we call *class profiling*, which basically groups possible value of these data properties (for our example applications, the image quality) into several classes, and profiles components with representative data in each class. Values between different classes are estimated using linear interpolation. Such class-based profiling provides a more accurate model of component behavior.

5.2 Problem Definition

An **augmented communication path**, $D = (c_1, \ldots, c_n)$, is a sequence of type-compatible components, in which c_i's output is sent to the input of c_{i+1}.

A **route**, $R = \{N_1, N_2, \ldots, N_p\}$, is a sequence of nodes separated by links . Each node N_i is modeled in terms of its *computation capacity*, $\mathrm{comp}(N_i)$ (operations per second), and a link between two nodes, $L_i = (N_i, N_{i+1})$, is modeled in terms of its bandwidth, $\mathrm{bw}(L_i)$. Both $\mathrm{comp}(N_i)$ and $\mathrm{bw}(L_i)$ are defined in terms of the shares of resources along the route available for a particular path.

A **mapping**, $M : D \rightarrow R$, associates components on augmented communication path D with nodes in route R. We are only interested in mappings that satisfy the following restriction: $(M(c_i) = N_u) \bigwedge (M(c_{i+1}) = N_q) \Rightarrow u \leq q$, i.e., components are mapped to nodes in path sequence order. The intuition behind this is that sending data back and forth between nodes along a route usually results in poor performance and wastes resources.

Our path creation strategies exploit the type compatibility described in the last chapter to identify valid composition patterns. The relation between types and com-

ponents is depicted using a **type graph** G_{t}: a vertex in the graph represents a type, and an edge represents a component that can transform data from the source type to the sink type.

The path creation problem can now be formally stated as the following: given a route R (with the resource shares allocated to the path), a type graph G_{t}, a source data type t_{s}, a destination data type t_{d}, select an augmented communication path D that transforms t_{s} to t_{d} and can be mapped to R so as to satisfy the following requirements:

- Type compatibility between adjacent components.
- Optimal performance. Performance can mean different things, for example, maximum throughput, minimal latency etc.

5.3 Overview of Our Solutions

Our path creation strategies automatically select and map a type-compatible component sequence to underlying network resources. In addition to satisfying type requirements, the strategies respect constraints imposed by node and link characteristics and optimize some overall path metric such as response time, data quality, or throughput.

We first describe a base version of the algorithm, based on dynamic programming, in which a single performance metric needs to be optimized (e.g. maximum throughput or minimal latency).

We then present an extension for applications that require the value of some performance metric to be in an *acceptable range*. For such applications, only after that range has been met does the application worry about other preferences. For example, most media streaming applications usually demand a suitable data transmission rate

(in some range) so that received data can be rendered appropriately at display devices; once the transmission rate is kept in that range, other factors such as data quality become the concern. We use the terms *range metrics* and *performance metrics* to refer to the two types of preferences.

Lastly, we describe a "local" scheme that can be used for a portion of the communication path. Using local planning, disjoint segments of a communication path can adjust their behaviors independently and concurrently while maintaining some overall performance guarantee. Such a local scheme can improve adaptation agility in that any portion of a communication path can be modified to respond to local changes in its network segment. More importantly, such schemes are indispensable for deploying path-based infrastructures in the situations where a communication path need to span multiple network domains, for which fine-grained coordination across different network domains is either prohibitively expensive or infeasible due to administration policies.

5.4 Base Algorithm

Unfortunately, finding the optimal solution for the path creation problem defined in Section 5.2 is an NP-hard problem.[1] The complexity mainly comes from the large number of possible composition among candidate components.

However, this problem can be made tractable with a reasonable simplification: we partition the computation capacities of nodes into a fixed number of *discrete* load intervals, i.e., capacity is allocated to components only at interval granularity. This

[1]The multiple choice knapsack problem can be converted to a simplified version of this problem

practical assumption allows us to define, for a route R, the notion of an *available computation resource vector*, $\vec{A} = (r_1, r_2, \ldots, r_p)$, where r_i reflects the available capacity intervals on node n_i (normalized to the interval [0,1]).

In the description that follows, we use maximum throughput as the goal of performance optimization. Note that the throughput is an application performance metric, i.e. the number of semantic units that pass through the network in a time unit, for example how many frames per second are delivered for a streaming application. We use p as the number of hosts in route R (i.e. $p = |R|$); m as the total number of types (i.e. $m = |V(G_{\mathrm{t}})|$); and n as the total number of components.

Dynamic Programming Strategy

The intuition behind the algorithm is to incrementally construct, for different amounts of route resources, optimal paths from the source data type to all types with increasing numbers of components, say $i + 1$, using as input optimal partial solutions involving i or fewer components.

To construct a solution with $i + 1$ (or fewer components) for a given type t and resource vector $\vec{A}$, we consider all possible intermediate types t' that can be transformed to t, i.e. all those types for which an edge (t', t) is present in the type graph. For each such t', consider all possible mappings of the associated component c on nodes along the route that use no more than $\vec{A}$ resources. Each mapping of c transforms the available resource vector to $\vec{A}'$ (after accounting for load times the incoming data volume), and provides a new mapping that combines this component with the previously calculated solution for t' with i (or fewer) components and resource vector $\vec{A}'$. The

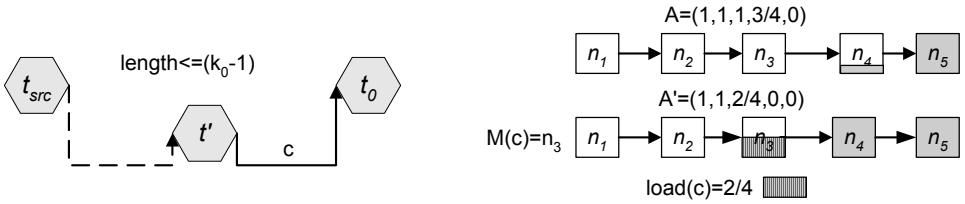


Figure 5.1: Map c to N_3 and lookup solution with $\vec{A}'$

combined mapping that yields the maximum throughput is deemed the solution at the $(i+1)^{\text{th}}$ level.

Because this procedure runs backwards from the destination to the source (i.e. c_{j+1} is mapped before c_j), consequently, only resource vectors of the form $(1, ..., 1, r_j \in [0, 1], 0, ..., 0)$ are used in the calculation. These set of such resource vectors is designated as RA. It is obvious that the size of RA is $O(p)$, where p is the number of nodes along the route.

Formally, the algorithm fills up a table of partial optimal solutions ($s[t_{\mathrm{s}}, t, \vec{A}, i]$) in the order $i = 0, 1, 2, \ldots$. The solution $s[t_{\mathrm{s}}, t, \vec{A}, i]$ is the data path that yields maximum throughput for transforming the source type t_{s} to type t, using i or fewer components and requiring no more resources than $\vec{A}$ ($\vec{A} \in \mathrm{RA}$). Figure 5.1 shows the moment in the calculation of $s[t_{\mathrm{s}}, t_0, (1, 1, 1, 3/4, 0), i+1]$ when component c is mapped to node n_3, and appended with partial solution $s[t_{\mathrm{s}}, t', (1, 1, 2/4, 0, 0), i]$. Note that in this example, computation capacity of nodes is partitioned into 4 intervals.

The algorithm is listed in Figure 5.2. Line 3 of the algorithm handles the base case: only the case with $t = t_s$ achieves non-zero throughput. Lines 8–13 represent the induction step, examining different drivers to extend the current partial solution for each specific intermediate type t and resource vector $\vec{A}$. Lines 12 and 13 ensure

Algorithm *Plan*
Input: $t_{\rm s}$,$t_{\rm d}$, $G_{\rm t}$, R
Output: The augmented path that yields maximal throughput from type $t_{\rm s}$ to $t_{\rm d}$ on route R

1. ($*$ Step 1: Initialization for partial plans with zero components $*$)
2. **for** all t, and $\vec{A} \in \mathrm{RA}$
3. **do** calculate $s[t_{\rm s}, t, \vec{A}, 0]$
4. ($*$ Step 2: Incrementally building partial solutions $*$)
5. **for** $i \leftarrow 1$ **to** pn
6. **do for** all $t \in V(G_{\rm t}), \vec{A} \in \mathrm{RA}$
7. **do** $s[t_{\rm s}, t, \vec{A}, i] \leftarrow s[t_{\rm s}, t, \vec{A}, i-1]$
8. **for** all $c = (t', t) \in E(G_{\rm t})$
9. **do for** all N_j with $\vec{A}[N_j] > \mathrm{load}'(c)$
10. **do** $M(c) \leftarrow N_j$
11. $\vec{A}' \leftarrow (\vec{A}[0], \ldots, \vec{A}[N_j - 1], \vec{A}[N_j] - \mathrm{load}'(c), 0, \ldots)$
12. $\mathrm{TH} \leftarrow \mathrm{throughput}(\mathrm{append}(s[t_{\rm s}, t', \vec{A}', i-1], c, \vec{A}))$
13. **if** $\mathrm{TH} > s[t_{\rm s}, t, \vec{A}, i]$
14. **then** $s[t_{\rm s}, t, \vec{A}, i] \leftarrow \mathrm{TH}$
15. **return** $s[t_{\rm s}, t_{\rm d}, \vec{A} = [1, 1, ..., 1], pn]$

Figure 5.2: Base Path Creation Algorithm

that the component achieving the maximum throughput defines the next-level partial solution. To optimize other performance metrics (e.g. shortest latency), only lines 12–14 of the algorithm need to be changed accordingly.

Table 5.1 shows how to calculate throughput of a communication path, i.e. the minimal value among the throughput of the nodes and links along the path. To calculate throughput values of individual links and nodes, compression ratios before links ($C(L_i)$) and components ($C(c_i)$) need to be calculated. These compression ratios represent the number of bytes, at those points (before L_i or c_i), generated by one byte of data at the source. So the throughput of a link is its bandwidth "decompressed" by the compression ratio. The division by the data unit size at source converts the

Path	$D = \{c_1, ..., c_n\}$
Route	$R = \{N_1, N_2, ..., N_p\}$, $L_i = (N_i, N_{i+1})$
Data unit size at source	S
Node Component Set	$M^{-1}(N_i) = \{c_{i_1}, ..., c_{i_m} \mid M(c_{i_k}) = N_i, 1 \leq i \leq k\}$
Accumulated Compression (link)	$C(L_i) = \Pi_j \mathrm{bwf}(c_j)$, for $M(c_j) = N_k$ with $k \leq i$
Accumulated Compression (driver)	$C(c_i) = \Pi_{j=1}^{i-1} \mathrm{bwf}(c_j)$
Throughput (link)	$\mathrm{TH}(L_i) = \mathrm{bw}(L_i)/(S \cdot C(L_i))$
Throughput (node)	$\mathrm{TH}(N_i) = \begin{cases} \mathrm{TH}(L_{i-1}) & \text{if} M^{-1}(N_i) = \emptyset \\ \frac{\mathrm{comp}(N_i)}{S \cdot \Sigma_{k=1}^{m} (\mathrm{load}(c_{i_k}) \cdot C(c_{i_k}))} & \text{if} M^{-1}(N_i) \neq \emptyset \end{cases}$
Path Throughput	$\mathrm{TH} = \min_i(\mathrm{TH}(L_i), \mathrm{TH}(N_i))$

Table 5.1: Calculation of throughput of a communication path

granularity from bytes to data units. Similarly, the throughput of a node, if there are any components residing on that node, is its computation capacity divided by the load for processing a data volume that corresponds to one data unit at the source. The expression $\Sigma_{k=1}^{m}(\mathrm{load}(c_{i_k}) \cdot C(c_{i_k}))$ represents the load at node n_i caused by one byte of data at the source.

One additional point about our algorithm needs some clarification: We need to know how much resources ($\mathrm{load}'(c)$ in line 9) to set aside for component c before we can combine c with an optimal Step $i-1$ solution. The problem here is that c's resource requirements $\mathrm{load}(c)$ are expressed in terms of per-input byte costs, and are difficult to evaluate without knowing what the input data volume is, which itself is only known once the Step $i-1$ solution is selected. Our solution to break this cyclic dependency is to first *guess* the resource requirements of c and then evaluate the throughput for this guess. Note that because of discretized load levels, we only need to make a constant number of guesses at each step, thus this does not change the

complexity of the whole algorithm.

The algorithm terminates at Step pn, with the solution in $s[t_{\mathrm{s}}, t_{\mathrm{d}}, (1, ..., 1), pn]$. This follows from the observation that there is no performance benefit from mapping multiple copies of the same component to a node. The complexity of this algorithm is $O(n^2mp^3) = O(n^3p^3)$[2] as opposed to $O(p^n)$ for an exhaustive enumeration strategy. n, the total number of components, usually is a big number. Even for a simple operation, such as compression, there may exist many different candidates, not to mention that each component may have multiple configurations. Therefore, $O(p^n)$ is infeasible in practice. In most scenarios, p is expected to be a small constant, therefore overall complexity of our path creation algorithm is determined by the number of components.

Two implementation issues need additional attention here. First, reducing the size of the type graph is important. When calculating paths, only types that can be reached from both source and destination types are considered. In addition, type ranks (described in Section 4.2.1) can be used to further reduce the size of type graph. These mechanisms help because of the observation that the total number of possible composable operations involving a specific type is limited. Second, when a type object needs to be made available across a network link, the augmented part of the type object needs to be calculated on the other side of the link using the link property transformation rules described in Section 4.2.

[2]It is safe to assume that $m < n$, i.e. the total number of types is less than the total number of components. This is because a type exists only if some components can produce and consume data of that type.

5.5 Extension 1: Planning for Value Ranges

As mentioned before, some applications require some performance metrics to be in an acceptable range, and only after the requirement of these range performance metrics is met, other performance metrics become interesting. To support such applications, given that our planning algorithm constructs data paths by incrementally filling in a solution table of $s[t_{\mathrm{s}}, t, \vec{A}, i]$, it is natural to extend this to check that retained solutions satisfy two conditions: (1) values of range metrics achieved by the current solution lie within the desired range, and (2) the value of any performance metrics is in fact optimized.

Although this is the basic idea of the extension, for some range metrics, such as path latency, additional work is needed. For such range metrics, even if the current value of the range metrics is not in the range for a partial solution, this does not exclude the possibility that this partial path may actually become a part of the final solution. For example, appending compression components to a partial path can bring down overall path latency by reducing packet size. So such candidates cannot be pruned.

To *estimate* whether the desired range can in fact be achieved by appending additional components, we employ a procedure called *complementary planning*, which just runs the planning algorithm in reverse, providing information about whether or not the range metrics can meet the requirement using residual resources from type t to t_{d}. In the process of *complementary planning*, component parameters are also *reversed*, modeling the situation where data flows from output ports to input ports. A reversed solution $s'[t_{\mathrm{d}}, t, \vec{A}' = (0, .., 0, r, 1, ..., 1), i]$ represents the communication

path with the best performance from t to t_{d} using resources less than $\vec{A}'$. When calculating $s[t_{\mathrm{s}}, t, \vec{A}, i]$, those partial solutions that can not meet the range requirement will be discarded by looking up $s'[t_{\mathrm{d}}, t, (1, ..., 1) - \vec{A}, pn - i]$. Heuristic functions are used for choosing among candidate paths that can all meet the required range. For example, data quality can be used to choose between two solutions that both can provide required throughput. Note that complementary planning needs to be run just once for the whole calculation.

5.6 Extension 2: Local Planning for Segments of the Network Route

The challenge in replanning for portions of an existing communication path is how to modify these portions independently while still maintaining some overall performance guarantee. For example, we would like to ensure that the range metrics for the entire path still fall within their desired range. Note that local mechanisms may compromise optimality of performance metrics, but we look at this as a reasonable tradeoff between global optimality and the benefits of local mechanisms as mentioned before, i.e., increased responsiveness to changes and eliminating the need for coordination across different network domains.

Our local planning strategy is a straightforward extension of the range planning mechanism described above. To create a partial path for R', which is a segment of the original route R, all we need to do is to run the range planning algorithm on R' with localized parameters. Since the types entering and leaving R' and the size of the incoming data units are known, the only thing left is to adjust the range metrics for

R'. Adjustment for throughput and latency is shown below:

- For applications that require an overall throughput range $[\mathrm{th}_{\mathrm{low}}, \mathrm{th}_{\mathrm{high}}]$, this can be done by assuring that each disjoint region in the path plans with the same range, which also gives them the most flexibility for building paths.
- For applications that require a latency range $[l_{\mathrm{low}}, l_{\mathrm{high}}]$, the localized latency range will be $[l_{\mathrm{low,R'}}, l_{\mathrm{high,R'}}]$, where $l_{V,R'}$ is the divided portion of latency l_V over segment R'. One way of doing this division is to consider the contribution of links in R' to the overall latency of R.

5.7 Distributed (Incremental) Planning

Though our path creation strategies have so far been described in a centralized way, they can easily be extended to run in a distributed fashion. To do that, each node (n_i) on the route calculates $s[t_{\mathrm{s}}, t, \vec{A} = (\underbrace{1, ..., 1}_{i}, 0, ..., 0), \sum_{j=0}^{i}(\mathrm{CN}_j)]$ where CN_j is the total number of components on node n_j.[3] In particular, the first node just needs to calculate its part of the solutions (for all possible types t) and send these partial solutions to the next node; upon receiving partial solutions from an upstream node, a node calculates its own solutions using the partial solutions from the upstream node. This procedure continues until it reaches the client node.

The primary benefit of this distributed version is that there is no need for a centralized planner, which requires a complete knowledge of components and types for all nodes in the route. By incrementally calculating a path in such a distributed fashion,

[3]The value of CN_j is not important for nodes other than n_j. Every node just tries to append partial solutions from the direct upstream node with as many of its own components as possible, CN_j is the maximal number of components that can be appended. Recall that $s[t_{\mathrm{s}}, t, \vec{A}, i]$ is the best solution using *no more than* i components.

only knowledge for common types that are used across different network domains is needed. A network node does not need to know which components exist on other nodes. This distributed version, combined with the local mechanisms described earlier make it much more practical for a path-based infrastructure to be deployed in the Internet, where a path usually spans multiple administration domains.

The extra traffic incurred for this distributed version is only messages of partial solutions (with $\vec{A} = (1, ..., 1, 0, ...)$) between adjacent nodes. It should be noted here that only values of the performance metric are needed, transmission of components and connectivity information is unnecessary. The size of such messages should be small since the number of possible common types is expected to be small along a communication path with fixed source and destination types. Heuristic strategies can be exploited to further reduce the number of partial solutions that need to be transmitted.

5.8 Summary

In this chapter, we have presented a model and algorithms for automatically constructing network-aware paths. Such paths can provide applications with required data and optimized performance for the underlying network conditions. Our strategies are very flexible in that they can work with applications with different performance requirements (maximum/minimum values or value ranges), and they can be used to construct an entire new path as well as modify portions of an existing path. The path calculation can be carried out by a centralized entity or incrementally from one network domain to another in a distributed fashion. The local planning and distributed

implementation are important for such path-based infrastructures to be used in a wide area network, where information exchange (about components etc.) between different network domains is either expensive or impossible due to administration constraints. The evaluation of our path creation strategies is described in Chapter 9.

Chapter 6

System Support for Efficient Path Reconfiguration

To cope with dynamic changes in the network, a network-aware communication path needs to reconfigure itself when the current configuration can no longer meet its performance requirements.[1] Our solution of low-overhead reconfiguration has two parts: (1) a set of simple rules placing slight restrictions on component behavior, and (2) a reconfiguration protocol that leverages these restrictions. Before we describe these two parts, we observe that there are two major challenges in dynamically modifying a communication path.

First, path reconfiguration should provide semantic continuity guarantee. Since components within an augmented communication path can transform data from one type to another, the conventional notion of continuity, i.e. in-order byte level delivery,

[1] This can be detected by either monitoring changes in resource availability of the path or by comparing the expected performance of the path with the value actually measured at run time.

can not be applied directly to this scenario. Instead, the continuity required by applications is at the granularity of semantic segments. A semantic segment here refers to a demarcatable application-specific unit of data in transmission, e.g., an HTML page or an MPEG frame. Conventional properties such as in-order transmission and exactly-once delivery can now be defined at the granularity of semantic segments.

Second, a path reconfiguration should avoid introducing a long interruption period in data transmission. To reduce reconfiguration overhead, mechanisms that can adapt to "local" changes in the network by modifying small portions of a whole communication path are important.

6.1 Reconfiguration Semantics

With the notion of semantic segment in hand, it is now possible to define what the application can assume about the received data after a portion of the communication path is modified. Our reconfiguration protocols can be customized to provide three different levels of semantics:

- **Level 1** semantics provides no guarantees, leaving it up to the application to reconstruct any lost data. This can be used for applications that involve non-critical data (e.g., news feeds), or applications that themselves can exploit in-order delivery guarantees to perform efficient recovery.
- **Level 2** semantics provides the guarantee of delivering complete *semantic segments*, essentially simplifying the task of the application recovery code. For example, in a streaming media application, a semantic segment might correspond to one video frame. Level 2 semantics ensure that a frame is either completely

delivered or not delivered at all.

- **Level 3** semantics provide full continuity guarantees with exactly-once semantics, completely isolating the application from the fact that the path has been reconfigured. Note that real-time applications can still detect a break in data availability; we take the view that such applications are best handled by inserting additional application-specific components that provide necessary timeliness guarantees. For an example of such components, see section 4.3.

6.2 Rules Restricting Driver Behaviors

To guarantee the above semantics, our framework relies upon the *semantic segment* and *soft state* properties of components in an augmented path, introduced in Section 3.2.

First, drivers are required to consume and produce data at the granularity of an integral number of semantic segments. Informally, this requirement ensures there must exist some points where data transmission can be safely suspended and path reconfiguration can be carried out with the semantic continuity guarantee. We will revisit this point later in the description of our reconfiguration protocols. Note that this property only refers to the logical view of the driver, and admits physical realizations that transmit data at any convenient granularity as long as segment boundaries are somehow demarcated (e.g., with marker messages).

Second, drivers are required to contain only soft state, which can be reconstructed simply by restarting the driver. Stated differently, given a semantically equivalent sequence of input segments, a soft-state driver always produces a semantically equiv-

alent sequence of output segments. For example, a `Zip` driver that produces compressed data will produce semantically equivalent output (i.e., uncompressed to the same string) if presented with the same input strings. This property ensures that drivers can be dynamically removed from or inserted into an existing path, and data retransmission can be used to reproduce the same output sequences.

Together, these two properties enable low-overhead path reconfiguration as described below.

6.3 Reconfiguration Protocol

The reconfiguration process is triggered by dynamic changes in the network, and is carried out by path control entities along the communication path. Path reconfiguration consists of three major steps: (1) generating a new plan, (2) ensuring required semantics prior to suspending data transmission, and (3) deploying the new plan and resuming data transmission.

Step 1 uses the planning algorithm described in Chapter 5, optionally reusing some of the solutions of previous calculations (e.g. by caching previously calculated paths), and can be overlapped with ongoing transmission. Steps 2 and 3 are controlled using our reconfiguration protocol.

Reconfiguration requires slightly different support for the three levels of reconfiguration semantics described earlier. Since required activities for Levels 1 and 2 are a subset of that for Level 3, our description focuses on the latter. The underlying problem is that in order to provide in-order and exactly-once semantics, any path reconfiguration scheme must take into account the fact that the portion of the path being

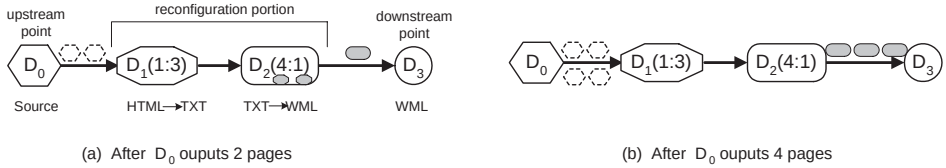


Figure 6.1: An example of data path reconfiguration using semantics segments.

reconfigured can have partially processed data either in the internal state of drivers or in transit across the network, or data that has been lost due to failures. Note that the soft-state requirement on its own does not provide any guarantees on semantic information loss or in-order reception.

Figure 6.1 shows an example highlighting this problem. To introduce some terminology, we refer to the portion of a communication path that needs to be modified due to changes in the network as the *reconfiguration portion*, and the components immediately upstream and downstream of this portion as the *upstream point* and *downstream point* respectively. In the example, driver d_0 is an HTML data source, and d_3 is a component receiving WML data. The reconfiguration portion consists of drivers d_1 and d_2. In this case, let's assume that driver d_1 converts every incoming HTML pages into three TXT pages, and driver d_2 composes every four incoming TXT pages into a WML deck. Consider a situation where system conditions change after the upstream point d_0 has output two HTML pages, and the downstream point d_3 has received one WML deck. At this point, the reconfiguration portion cannot be replaced because doing so affects semantic continuity. It is incorrect to retransmit either the second page from d_0 whose effects have been partially observed at d_3, or the third page, which would result in a loss of continuity at d_3.

Our reconfiguration protocol leverages the semantic segments and soft state restrictions placed on driver functionality as follows.

Intuitively, the first restriction allows us to infer which segments arriving at the downstream point of the reconfiguring portion depend on a specific segment injected at the upstream point and vice-versa, while the second makes it always possible, even if any internal driver state is reset, to recreate the same output segment sequence at the downstream point by just retransmitting selected input segments at the upstream point.

The basic idea of our solution is to delay the reconfiguration to *safe points* in data transmission where the reconfiguration portion can be safely removed, and semantic continuity can be achieved using *selective retransmission* of data that has not been seen downstream of the reconfiguration portion.

The key to detecting these "safe" points is to keep track of the correspondence between segments received at the downstream point and the segments sent from the upstream point, which is determined by the driver characteristics in the reconfiguration portion. If a reconfiguration portion contains a sequence of drivers $D = \{c_1, ..., c_n\}$ of which driver c_i produces p_i semantic segments upon receiving q_i input segments, we refer to $p/q = \Pi_{i=1}^{n} p_i/q_i$ as the *synthesis factor* of the reconfiguration portion (here p and q are relative primes). For the reconfiguration portion, the semantic information in the jth outgoing segment from the upstream point is contained in segments within the range of $[\lfloor (j-1) \cdot p/q \rfloor + 1, \lceil jp/q \rceil]$ received by the downstream point. More interesting is the fact that the boundary of each $(i \cdot q)$th segment at the upstream point is preserved at the downstream point, which corresponds to the boundary of the $(i \cdot p)$th segment. This means that after the downstream point receives such a segment, all seg-

ments (inclusively) before the $(i \cdot q)$th segment must have been seen at the downstream point and there is no state of these segments left in the reconfiguration portion. Such segments are referred to as *flushing* segments in our reconfiguration protocol to reflect the fact that these segments can in effect completely push state (and data) remaining in the reconfiguration portion (of previous segments) to the downstream point.

Note that in practice p_i/q_i may not necessarily be a constant number, so our framework exploits a flexible mechanism that tracks these flushing segments by using marker messages, which demarcate segment boundaries. All drivers along a communication path are required to pass only incoming markers that match their output segment boundary (others will be discarded). Therefore, receipt of a marker at the downstream point of a configuration portion signifies the end of a flushing segment (at the upstream point).

6.3.1 Reconfiguration Process

The state diagram of a path during a reconfiguration is depicted in Figure 6.2. In this figure, a bold font is used for distinguishing control messages or data segments from actions taken by the path (shown in italics). The reconfiguration process includes the following steps.

1. Upon receiving a message signifying the start of a reconfiguration (with the new plan) , the downstream point starts to monitor incoming data ($1'$) (the `Monitor` state); the upstream point starts to buffer outgoing segments while continuing to deliver them downstream (1)(the `Buffering` state). Besides, a marker is appended at the end of each output segment from the upstream point. Other

nodes within the reconfiguration portion do not change their state.

2. The downstream point continues monitoring until it receives a marker from the upstream point, which signifies the end of a flushing segment from the upstream point. The downstream point then sends a `Seg_Ack` message to the upstream point, and begins to discard any further incoming data segments (2).

3. Upon receipt of the `Seg_Ack` from the downstream point, the upstream point suspends (keeps buffering but not delivers data downstream) data transmission and sends a `Modify` message to all nodes that are involved in the reconfiguration (3).

4. Upon receipt of a `Modify` message, all nodes in the reconfiguration portion enters the `Recfg` state, tearing down the components in the old configuration and replacing them with the new component graph. In this stage, all drivers within the reconfiguration portion except the upstream point discard any incoming data (4'). The upstream point continues buffering outgoing segments (4).

5. After the modification on a node is finished, an `ACK` message will be sent to the upstream point (5).

6. After receiving `ACK` messages from all nodes, the upstream point resumes data transmission, starting with retransmission from the segment that follows the last flushing segment received by the downstream point (6). Note that since every driver is associated with a unique ID, new components after the reconfiguration will not accept data from the network that is addressed to the old components.

The process described above achieves semantics level three reconfiguration semantics. For semantics level two, the data buffering and retransmission actions can be omitted. For semantics level one, step 1 and 2 can be further bypassed, i.e., the upstream point needs not wait for the **Seg_Ack** message from the downstream point.

Example For the example shown in Figure 6.1, reconfiguration works as follows. First, the upstream point (d_0) starts buffering every segment it produces after the reconfiguration begins. The downstream point (d_3) will receive a marker after the third page from d_2, which is the marker appended at the end of the fourth page from the upstream point. It then sends an acknowledgement to the upstream point. After that, data transmission will be suspended at d_0 so that d_1 and d_2 can be replaced with another compatible driver graph. To resume data transmission, d_0 retransmits buffered data starting from the fifth page.

6.3.2 Error Recovery

In addition to adapting to changes in resource availability, our scheme can also be used for "extreme" cases where link or node errors cause lost of data or driver state. The only difference between the two situations is whether the reconfiguration protocol is executed on demand or runs all the time.

To gracefully recover from the failures of links and nodes along a communication path, we need to do buffering and monitoring all the time at the upstream and downstream points of an unstable network segment. Moreover, the downstream point needs to delay the delivery of received segments until it receives a marker from the upstream point. Meanwhile, the downstream also needs to send acknowledgements

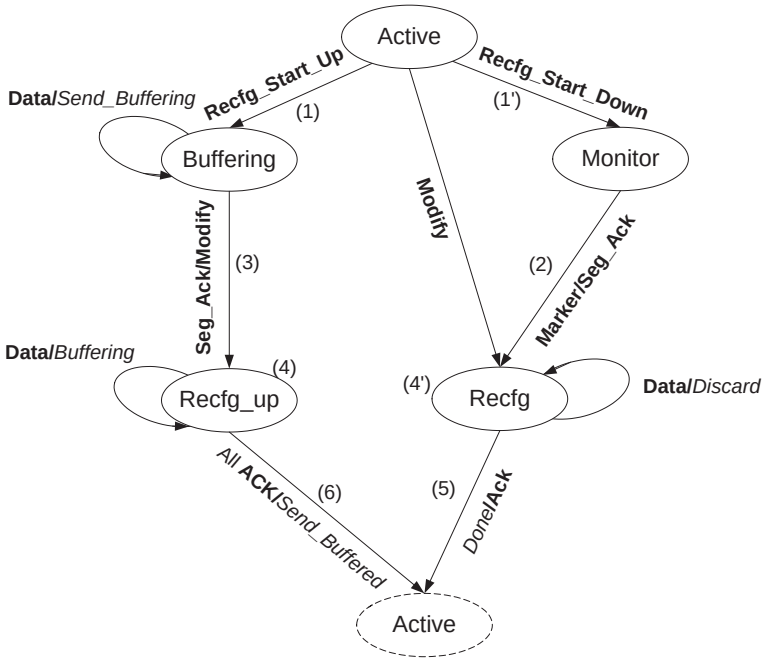


Figure 6.2: State diagram of path reconfiguration. Numbers on arcs correspond to the steps described in the text.

of received markers to the upstream point so that the upstream point can free buffer space accordingly. Upon recovery from a network failure, the downstream point discards its buffered data and resends the acknowledgement of the last received marker to the upstream point. The procedure that follows is exactly the same as steps (3) –(6) in the reconfiguration process described earlier.

6.4 Local Reconfiguration

In addition to modifying a whole communication path (where it called *global reconfiguration*), the reconfiguration process can also be applied to allow individual nodes

or small portions of a communication path to adjust their behaviors independently and concurrently. We refer to the latter as *local reconfiguration*. By using local reconfiguration, every segment of a communication path can independently and concurrently adapt to dynamic changes in the network. This not only results in better responsiveness, but more importantly, also enables each network domain along the path to control its portion independently, especially important for infrastructures that run in a wide area network, such as the Internet.

To support local reconfiguration, in addition to the reconfiguration process described earlier, we need two more things. First, we need a planning algorithm suitable for generating a small path portion to replace a part of an existing communication path while retaining some overall performance guarantee. In Section 5.6, we have described an algorithm that can provide such support. Second, we need strategies to determine which part of a communication path (i.e. nodes and links) should be involved in a local reconfiguration. In this section, we focus on this issue.

To start with an example, if the bandwidth of a network link changes, local reconfiguration may first try using only the direct upstream node of that link. If the new calculated plan can cope with the change, it will be deployed without further action. Otherwise, the reconfiguration portion has to be increased to involve more network resources until the situation is handled. Note that this propagation can be terminated at any time by just invoking a global reconfiguration.

The tradeoff in choosing an appropriate point to switch between local and global reconfiguration involves the length of the segment selected for reconfiguration (which affects reconfiguration cost), and the likelihood that the reconfiguration can successfully handle changes. Our framework uses a three-level strategy. Upon a reconfigura-

tion request, the first reconfiguration attempt happens in a single node when its load changes or the load on its direct downstream link changes. If the first reconfiguration attempt cannot meet the application requirements, then the second reconfiguration attempt will be triggered, which includes network segments comprising nodes connected with relatively fast links (usually within a single network domains). Lastly, if both of these fail, a global reconfiguration will be started for the whole communication path.

From a performance perspective, local reconfiguration can be viewed as a mechanism for balancing the tradeoff between adaptation agility and the optimality of data transmission. In a dynamic network environment where dynamic change is frequent and modification of the whole data path may introduce big overhead, we believe such mechanisms are desirable.

More importantly, mechanisms that allow individual segments to be constructed (see section 5.7) and modified in a distributed fashion can greatly reduce the need for global knowledge of components and resource availability, and coordination across multiple network domains. These aspects, in addition to the overhead of coordination across different network domains, are the most troublesome problems in managing long communication paths that span multiple network domains. Combining local reconfiguration with distributed path creation together, our approach allows *every network domain to have complete control over the path portions within its region without requiring coordination from others*.[2]

[2]For such cases, global path reconfiguration can be completely substituted by the combination of local reconfiguration and rebuilding a new path in a distributed fashion when local reconfigurations can not handle the change.

6.5 Summary

In this chapter, we have presented system support for modifying augmented paths in our framework with semantic continuity guarantees. Our scheme relies on a set of simple rules on component behaviors and an efficient reconfiguration protocol. In addition to modifying communication paths, our scheme can also be used for recovering data transmissions from failures of network nodes and links along a communication path. Moreover, our scheme can be used over a whole communication path (i.e. global reconfiguration) as well as with disjoint segments of a communication path independently and concurrently (i.e. local reconfiguration). Support for local reconfiguration can not only considerably improve the responsiveness of communication paths to dynamic changes, but also greatly eliminate the requirements of global knowledge and coordination across different network domains, which allows our framework to be used in a large scale network.

The overhead of path reconfiguration and the relative benefits of global and local reconfiguration are examined with experiments in Chapter 9.

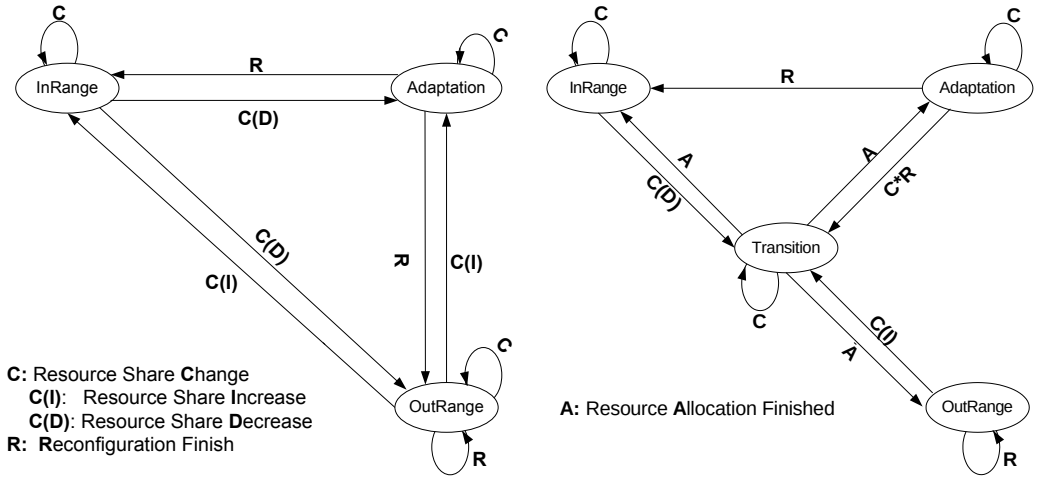
Chapter 7

Resource Management for Path-Based Infrastructures

For a path-based infrastructure to support network-aware paths in a wide area network, resource management is important for providing individual paths with optimized performance while maintaining a high throughput for the whole network (i.e. accepting as many paths as possible). In particular, there are two questions that need to be answered:

- How should shared network resources be allocated among multiple paths that pass through them?
- How should computation resources be distributed among different network regions in order to achieve better overall performance?

In this chapter, we present our solutions for these questions.



(a) (b)

Figure 7.1: (a) State Transitions for a Network-Aware Communication Path (b) In Our Scheme.

7.1 Resource Sharing among Multiple Paths

We start by describing a strategy that answers the first question posed at the start of this Chapter, namely how should shared network resources be allocated among multiple paths that pass through them. The goal is to ensure that optimized performance is delivered to as many paths as possible.

To solve this problem, we first need to understand how a network-aware path behaves within a shared network environment. Figure 7.1(a) shows the state transitions of such a path during its lifetime (the start and finish states are omitted to simplify the presentation). If the resources allocated to the path are sufficient for it to meet the performance requirements, a path is deemed to be in the `InRange` state, i.e., its performance is in the desired range. When some of its resource shares change, there are two possibilities: either it continues to meet its performance requirements or not.

In the latter case, there are also two possibilities depending on whether or not the path can manage to go back to the InRange state by reconfiguring itself. If it can, it stays in a state called `Adaptation` until reconfiguration is complete. If not, it enters the `OutRange` state, from which it can transit to the other states only after the path's allocated resource shares are raised. We call `InRange` and `OutRange` as *stable* states.

Examining the life time of an individual path, which is depicted in Figure 7.1(a), one can observe that there are three different types of resource shares that can be associated with the path. The first type is the share values used in planning for a new configuration (for path creations or reconfigurations). In general, the greater the value, the better the generated plan will be. The second type is the upper bound values of resource shares that the path is allowed to use, i.e. the allocated shares. The third type is the actually used shares by the path at a particular time.

Taking these three types of resource shares into consideration, we observe that in order to provide optimized performance to as many paths as possible, an ideal scheme for allocating resource shares among multiple paths should 1) maximize the value of the resource share for planning purposes to produce as good as possible a configuration; 2) minimize the difference between the allocated and the actual used shares to avoid resource waste.

This is the basic idea of our scheme, which employs the following two strategies: 1) when planning is needed, a path can increase its allocated shares by sending requests to all the resources involved; 2) whenever a path enters a stable state, it is required to release unused resources. The state diagram of an individual path using this scheme is shown in Figure 7.1(b). The *Transition* state is for a path to send *allocation*

requests to all the resources along the path. Upon receipt of an *allocation* request, a network resource is required to do its best to increase the allocated share of that path.

In addition to producing better path configurations, another benefit of using such *allocation* requests is that they can effectively balance load across the network. For example, if path A uses nodes n_1 and n_2, and among them n_1 is heavily loaded but n_2 has many unused computation resources. After sending *allocation* requests to both of them, path A will receive a larger share from n_2. As a result, A will place most of its computation on n_2. If the load in thhe network changes afterwards so that n_2 becomes the overloaded one and n_1 has unused resources, the same requests can effectively move computation required by A from n_2 to n_1. Though one can also achieve a similar effect through one tight coupling between n_1 and n_2, such cooperation usually requires expensive information exchange about dynamic resource availability, thus does not scale well for large networks.

To manage allocated shares of a path, our solution has two parts: (1) *allocation*, which determines the new allocated share in response to an *allocation* request; and (2) *adjustment*, which changes shares of existing paths, triggered either by a need to satisfy allocation requests from other paths or by releases of resources from other paths that have left or entered a stable state.

7.1.1 Allocation

The goals of our allocation strategy are to provide the largest possible allocated share (up to a maximum value, MAX) to ensure success during planning, while at the same time avoiding frequent reconfigurations and cascading adjustments. The algorithm listed in Figure 7.2 reflects these ideas.

Algorithm *Allocation*
Input: Path
Output: Allocated Share for the path
1. **if** $\mathrm{available} > \mathrm{MAX}$
2. **then return** MAX
3. ($* \ p$: number of paths, n: increase in p within the last time unit $*$)
4. $r \leftarrow \max(1, n)$
5. $p_r \leftarrow \max(\lceil p/r \rceil \times r, p + c)$
6. **if** (Path is a New Path)
7. **then return** $1/p_r$
8. **else return** $\max(\mathrm{current\_share}, \min(\mathrm{available}, 1/p_r))$

Figure 7.2: Calculation of the Value of the Allocated Share

When the resource is underutilized, allocation requests result in a predefined MAX amount of resources being allocated. Information about this amount can either be provided by the path or specified by the resource. Note that since paths return unused resources, allocating a large share for planning purposes does not negatively impact resource availability for future paths.

The case where the resource is oversubscribed (i.e., fewer than MAX resources are available) is more interesting. Intuitively, the algorithm implements a fair policy: the resource is equally partitioned among all active paths. However, this base policy needs to be refined to meet our original goals, namely to avoid frequent reconfigurations and cascading adjustments.

A situation where frequent reconfigurations happen with the base policy is when new paths are continually entering the system and making allocation requests. If paths were allocated a share of $1/p$ (where p is the number of paths) the arrival of each new path would force an adjustment of the shares granted to all existing paths, resulting in an undesirable user experience. To take this into account, our algorithm

"damps" the effect of path arrivals by instead allocating a smaller share $1/p_r$, where the quantity p_r is computed in terms of two parameters n and c as shown in Figure 7.2. As computed in lines 4–5, p_r takes on a new value only once for each time unit (using n, a prediction for the expected increase in the number of paths over the time unit). This has the benefit that each existing path would need to be adjusted at most once over a time unit.

Line 5 also shows how the parameter c is used to bound the minimum value of p_r. By observing that each adjustment of a path returns a share equal to $(1/p - 1/p_r)$, it follows that the fraction of paths that will need to be adjusted to grant an allocation request is $1/(p_r - p)$. The value of this fraction is at most $1/c$, thereby limiting the amount of work that will need to be done in the worst case. c would typically be a different predefined constant for each resource. In our experiments, we choose c to be 5% of the maximum number of paths that can be sustained on the resource.

The other refinement over the base scheme is shown in lines 6–8 of the algorithm, where different shares are returned depending on whether the allocation request is made by an existing path or a new one. For existing paths, the algorithm ensures that any increases in share allocation are constrained from above by the amount of available resources (i.e., those resources that can be granted without adjusting share allocations for other paths). To understand this policy, consider what would happen in its absence for a path A, which shares the resource r_1 with path B. If A requests an increase in its allocated share, the share of path B may need to be reduced. To maintain the required performance, path B may in turn issue allocation requests to increase its shares on other resources along the path. These requests from B may affect path C in a similar fashion if B and C happen to use the same resource r_2.

The same thing can happen for path D if C and D share another resource r_3, and so on. The situation would be even worse if D is actually A and r_3 is actually r_1, for which A initially tried to increase its share on r_1 but end up with a decreased share. In summary, such propagation may cause the whole network to oscillate with overwhelming allocation requests and adjustments. The constraint in line 8 is used to avoid this problem.

The algorithm in Figure 7.2 treats all paths uniformly for resource allocation purposes. Note that it is straightforward to extend this scheme to handle cases where some paths have higher priority than others by associating weights with paths.

7.1.2 Adjustment

Adjustment on allocated shares of existing paths occurs in two situations: (1) when there are insufficient resources available to satisfy an allocation request, and (2) when an allocated share is released.

For the first situation, a set of existing paths needs to be selected and their shares will be reduced in order to accumulate a large-enough share for the allocation request. The allocation step described earlier is responsible for determining how large this share needs to be; the adjustment step decides which paths to take away resources from. Several different heuristic schemes can be employed to guide the latter process. Our scheme picks victims in decreasing order of the allocated shares, affecting paths that have a larger share of the resource. This basic scheme can be extended to restrict attention only to paths in the `InRange` state. The intuition here is twofold. First, such paths are more amenable to reconfiguration for staying in the desired range, as opposed to the paths in the other states. Second, if resources are overcommitted, it is

usually more acceptable to reject new connections rather than push existing paths into the `OutRange` state. Note that, employing this extension may end up reducing the overall share that can be allocated to the requester.

The adjustment in the second situation is simpler: when a share of a resource is released, it is used to increase the allocation of paths in the `OutRange` state to the preconfigured maximum. The intuition here is the same as in the allocation step: providing paths with the maximal opportunity of reentering the `InRange` state. If no such path exists, a variety of heuristics can be employed. For instance, if the load on the resource is increasing (as observed by monitoring the number of active paths), the release resources are reserved for future connections; otherwise, the released shares are used to increase the allocation of the path with the smallest value of the allocated share.

7.2 Resource Distribution across Network Regions

We now present our solution for the second question posed at the beginning of this Chapter, i.e., how should computation resources be distributed among different network regions in order to achieve a better overall performance? For a path-based infrastructure, different distributions of a same amount of total computation capacity among the network nodes can result in very different performance. The goal is to improve the overall performance of the whole network.

Our strategy is motivated by the observation that although path-based infrastructures can in general deploy operators on any network node along a communication path, usable nodes in practice are most likely a small set of strategic nodes such as ISP

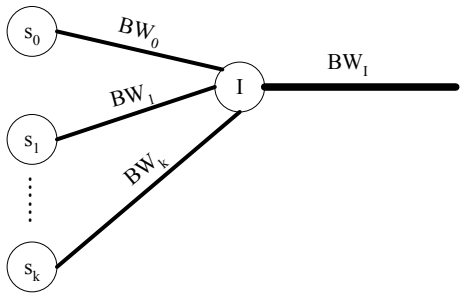


Figure 7.3: Hierarchical arrangement of servers and ISP nodes.

and gateway nodes. Besides, there usually exist some forms of administrative agreements between a higher-level network domain (e.g., the ISP) and a lower-level one (e.g., the server). Combining these two together, one can view the computation distribution problem as one of rearranging computation resources in a hierarchical network graph. Specifically, given a fixed *computation resource budget*, initially assumed allocated to nodes of a lower-level domain, the problem becomes one of moving a portion of the budget to nodes in a higher-level domain so that the overall performance of the whole network can be improved. The reason that such rearrangements result in better performance of the overall network is basically because of resource sharing; after such a rearrangement, overloaded servers can take advantage of shared resources at a high-level node in the network graph, contributed by servers that have a relative light load.

Our strategy aims to move the maximal amount of resources to high-level network domains with the guarantee that the performance of those domains, from which the computation resources are moved out, will not be compromised after the rearrangement.

This rearrangement problem is illustrated in Figure 7.3. Initially, each server s_i

has a computation budget C_i (number of operations per second), and is connected to the ISP node (I) via a link with bandwidth BW_i. The ISP node in turn is connected to a higher-level network domain with a link of bandwidth BW_I. We use the terms *server link* and *ISP link* to distinguish between the two types of links. We can further assume that[1]

$$\sum_i \mathrm{BW}_i \geq \mathrm{BW}_I \tag{7.1}$$

The problem is to determine what portion of C_i ought to be moved from s_i to I (and what portion of the computation resources at I can be moved to the higher-level...). In the description that follows, we first focus on how to distribute computation resources between these two levels. How to apply our strategy recursively within a network graph is deferred to the end of our description.

7.2.1 Algorithm for Distributing Computation Resources

In the context of a two-level network structure, the question that our strategy answers is what fraction of the computational resources from which servers can be transferred to the ISP node (see Figure 7.3) *without compromising* the performance of the contributing servers, namely leaving unchanged the number of connections that they can serve.

To describe our strategy, we need to introduce a model for client connection requests. We assume that all communication paths require a throughput of TH data units per second, and that these paths are of two kinds: *uncompressed* and *compressed*. The latter involves transcoding and/or compression operators to reduce its bandwidth re-

[1] Otherwise the bandwidth in the higher-level network domain would remain underutilized

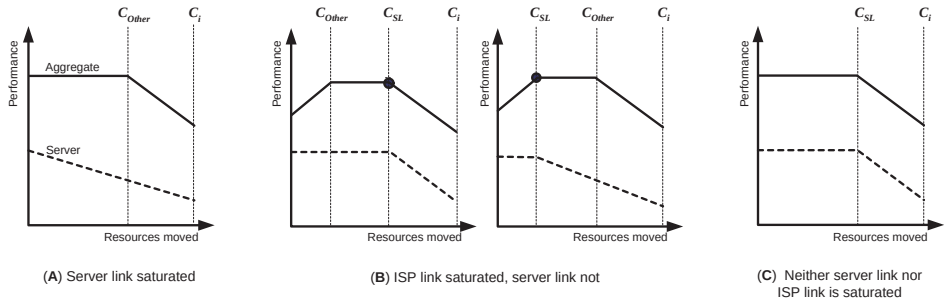


Figure 7.4: Performance impact of incrementally transferring computation resources from a single server node to the ISP node for a fixed load level. The three cases correspond to different saturation situations for the server and ISP links. C_{Other} denotes the maximum resource level that can be utilized for improving the performance of other servers. C_{SL} denotes the resource level at which the server link gets saturated.

quirement. Each compressed path requires an average computation of c operations per data unit,[2] and reduces bandwidth requirement by the fraction D. We further assume that the fraction $p_i (0 \leq p_i \leq 1)$ of all requests via ISP I is for accessing contents on server s_i, which has a computation budget of C_i. As shown in Figure 7.3, we refer to the bandwidth on the link connecting server s_i to I (the server link) as BW_i and that on the link connecting I to the Internet (the ISP link) as BW_I. Our strategy calculates C'_i, the computation resource left at each server s_i. Note that $0 \leq C'_i \leq C_i$.

Assuming the above client traffic pattern, the strategy identifies servers for whom the corresponding server link has unused capacity for load levels where the ISP link is operating at capacity. The rationale for this choice can be seen by examining Figure 7.4, which depicts, for a given load level, the impact on overall performance as resources are incrementally transferred from a particular server, s_i, to the ISP node. Depending on whether the ISP link is saturated or not, and whether a particular server

[2]Here we only consider the computation capacity required for manipulating data; the overhead of reading the content from disk and passing it through a protocol stack are not counted since these overheads are always present.

link is saturated or not (at the beginning point), one can distinguish three classes of server behavior: (A) when the server link is saturated; (B) when the server link is unsaturated while the ISP link is saturated; and (C) when neither the server link or the ISP link are saturated. For each class, Figure 7.4 shows the impact on aggregate system performance (solid line) and individual server performance (dashed line) as resources are incrementally moved out of the server.

When the server link gets saturated (case (A)), any movement of computation resources out of s_i will decrease the number of connections sustainable at the server. This decrease is offset at the aggregate system level until C_{Other} resources have been transferred, by other servers benefiting from the pooled resources.

When the ISP link gets saturated before the server link (case (B)), there are two situations. Both situations start off by seeing an increase in aggregate performance because of additional compressed connections being served on other servers. Meanwhile, moving computation resources out of s_i increases the bandwidth consumption at its server link but does not affect its performance. This situation continues until we reach a point where either there is no further benefit from additional ISP resources (the left figure), or the server link gets saturated (the right figure). In the first situation, aggregate performance levels off until the C_{SL} level is reached at which point both server and aggregate performance start decreasing. In the situation where the server link gets saturated first, server performance starts decreasing immediately but its impact on aggregate performance is offset as in case (A) above until the C_{Other} level is reached. The points marked by black circles in the case (B) figures represent the maximal amount of computation resources that can be moved out of s_i without degrading its performance.

$n_I = \mathrm{BW}_I/\mathrm{TH} + (1-D) \times \sum_i \frac{C_i}{\mathrm{TH} \times c}$ $\left[\begin{array}{c} n_I = n_c(\mathrm{compressed}) + n_{\mathrm{uc}}(\mathrm{uncompressed}) \\ n_c = \sum_i \frac{C_i}{\mathrm{TH} \times c} \\ n_{\mathrm{uc}} \times \mathrm{TH} + n_c \times D \times TH = \mathrm{BW}_I \end{array} \right]$	Maximum number of connections sustainable over the ISP link
$n_{\mathrm{sl},i} = \mathrm{BW}_i/\mathrm{TH} + (1-D) \times \frac{C'_i}{\mathrm{TH} \times c}$	Maximum number of connections can be sustained at the server link of s_i after moving some of its computation resources to the ISP node.

Table 7.1: Calculation of the Number of Connections Sustainable at ISP Link and Server Link.

In Figure 7.4(C), neither the ISP link nor the server link is saturated, so moving computation resources from s_i does not increase the aggregate performance unlike in case (B). This is understandable because at this time the number of connections that can be sustained at a server is unaffected by the amount of computation resources at the ISP node. Only after the server link gets saturated does both the aggregate and server performance decrease. Changes in load level can convert a case (C) server into either case (A) or (B) depending on whether the server link or the ISP link gets saturated first, thus this case can not be used to determine which server and how many computation resources can be moved out.

In light of the above analysis, our strategy restricts itself to identifying servers that would fall into category (B) above. For such servers, it is safe to move resources up to the point marked by the black circles in Figure 7.4(B) irrespective of the encountered load levels.

Servers whose server links remain unsaturated when the ISP link is saturated can be identified by comparing $p_i \times n_I$, the maximum number of server connections that can be sustained assuming that the ISP link becomes the bottleneck (p_i is the fraction

of connections directed towards s_i) with $n_{\mathrm{sl},i}$, the maximum number of server connections that can be sustained assuming the server link becomes the bottleneck. Table 7.1 shows how these parameters are computed by considering the number of compressed and uncompressed connections that can be supported by a given amount of computation and bandwidth resources. The $n_{\mathrm{sl},i}$ expression assumes that C_i' resources are left behind at the server. Servers in case (B) must have $n_{\mathrm{sl},i} \geq p_i \times n_I$, i.e. satisfy the following equation:

$$\begin{cases} 0 \leq C_i' \leq C_i \\ \mathrm{BW_i/TH} + (1-D) \times \frac{C_i'}{\mathrm{TH} \times c} \geq p_i \times n_I \end{cases} \tag{7.2}$$

It is easy to prove, by contradiction, that there must be at least one category (B) server. Let us assume that no server has a valid solution for Equation 7.2. This implies that (summing up over all servers)

$$\sum_i \frac{\mathrm{BW_i}}{\mathrm{TH}} + (1-D) \times \sum_i \frac{C_i}{\mathrm{TH} \times c} < \sum p_i \times n_I$$

The right hand side of the above inequality is just n_I, which in turn can be substituted by the corresponding expression from Table 7.1. Thus, our assumption leads us to the inequality

$$\sum_i \frac{\mathrm{BW_i}}{\mathrm{TH}} + (1-D) \times \sum_i \frac{C_i}{\mathrm{TH} \times c} < \frac{\mathrm{BW}_I}{\mathrm{TH}} + (1-D) \times \sum_i \frac{C_i}{\mathrm{TH} \times c}$$

This requires $\sum_i BW_i < BW_I$, which is in contradiction with our previous assumption of $\sum_i BW_i \geq BW_I$. Therefore, there must be at least one category (B) server.

The recursive algorithm employed by our strategy is shown in Figure 7.5. Lines 3–8 check, for a given load distribution, whether a server has a valid solution for Equation (7.2). If not, it is excluded from further consideration, with the available ISP link

Algorithm *Distribute*
Input: Server Set S, BW_I
Output: Distribution of computation between servers and the ISP node
1. $S' \leftarrow S$
2. $\mathrm{BW}'_I \leftarrow \mathrm{BW}_I$
3. **for** all $s_i \in S'$
4. **do if** for s_i equation (7.2) has a solution
5. **then** set C'_i
6. **else** $C'_i \leftarrow C_i$
7. $\mathrm{BW}'_I \leftarrow \mathrm{BW}'_I - BW_i$
8. $S' \leftarrow S' - s_i$
9. **if** $S' \neq S$
10. **then** Adjust load distribution for all $s \in S'$
11. **Call** *Distribute*(S',BW'_I);
12.

Figure 7.5: Distribution of Computation Resources between ISP and Server Nodes

bandwidth adjusted as shown in Line 7. To understand this, note that the bandwidth contribution of such servers on the ISP link cannot exceed BW_i, because no additional connections (compressed or uncompressed) for this server can be supported once the server link is saturated. The recursive call uses this reduced value of available ISP link bandwidth and adjusted load distribution values (based on the relative contributions from remaining servers). Note that the two invariants about load distribution ($\sum_i p_i = 1$) and bandwidth ($\sum_i \mathrm{BW}_i \geq \mathrm{BW}_I$) hold after each adjustment. The algorithm terminates when all servers in S' have valid solutions for Equation (7.2). It is only these servers that can contribute a portion of their computation budget to the ISP node. The amount that can be transferred is easily determined by picking the minimum value C'_i for each such server that still results in Equation (7.2) being satisfied.

Figure 7.6 illustrates this algorithm using an example configuration consisting of

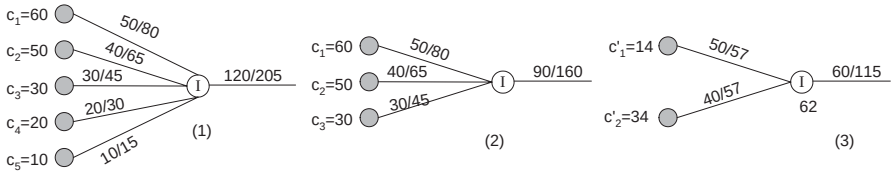


Figure 7.6: An example showing recursive calculation of the computation budget transferred to the ISP node.

5 servers with computation budgets 10, 20, 30, 50, and 60 units, with client connection paths requiring $\mathrm{TH} = 1$, $c = 1$, and $D = 0.5$. Client requests are uniformly distributed amongst these servers. The first call to the *Distribute* routine results in servers 4 and 5 being removed from S' because their server links cannot sustain $\frac{205}{5}$ connections. The second call removes server 3 because it cannot sustain $\frac{160}{3}$ connections. The algorithm terminates on the third call when both servers 1 and 2 can sustain $115/2$ connections, and can contribute a portion of their computation resources (amounts shown in the figure) to the ISP node.

Our description above considered a two level hierarchy (i.e. between servers and ISPs). This strategy can be easily extended to a hierarchically organized network domain with multiple levels. The basic idea is as follows: when moving resources to a n^{th} level node, count the resource budget of the $(n-1)^{\mathrm{th}}$ level node as the aggregate value of the resources at all levels (inclusively) lower than $n-1$; but only resources on the $(n-1)^{\mathrm{th}}$ level node (no lower level) can be moved to higher-level nodes.

A practical note: the algorithm sketched above assumes prior knowledge of the load distribution among low-level network domains. Since the load distribution varies over time, the redistribution process can be made more conservative by capping the maximal amount of resources that can be moved.

7.3 Summary

In this chapter, we have presented strategies for managing network resources for path-based infrastructures. This contains two parts. The first part is for allocating resources among multiple active paths on a shared network resource. We proposed a scheme that enables an individual path to achieve a better configuration by allocating it a share as large as possible, and allows the whole network to sustain as many as possible paths by reducing resource waste. Another good property of this scheme is that resource allocation is conducted in a completely distributed fashion without requiring information exchange on resource availability.

The second part is for distributing computation resources across the network to improve the overall performance of the whole network. Based on the current organization of the Internet, we proposed a hierarchical model and corresponding algorithm to rearrange computation resources by moving some portion of computation resources from lower-level nodes to higher-level ones in the network graph. Our scheme is able to set up maximal resource pools across the network, with the guarantee that the performance of network domains where computation resources are moved out will not be compromised after such rearrangements. The benefit of such rearrangements is that by setting up shared resource pools at high-level network nodes, overloaded network domains can always take advantage of spare resources contributed by other domains. We evaluate these two strategies in Chapter 9 by simulating their effect in the context of a large network supporting many simultaneous client requests to media servers.

Chapter 8

Implementation: CANS Infrastructure

We have implemented a prototype of our framework in the form of a programmable network infrastructure, called **C**omposable **A**daptive **N**etwork **S**ervices (CANS in short). CANS contains implementations of all the schemes described in Chapter 3 to Chapter 7.

The kernel of the CANS infrastructure is the CANS Execution Environment (EE). The CANS EE serves as the runtime system for components in augmented communication paths, and provides all the infrastructural support required by these paths to realize network-aware data communication: delivering data across networks, managing resources and communication paths (i.e. path creation and reconfiguration), downloading mobile code, and providing resource availability information.[1] A CANS net-

[1]The current implementation of CANS relies upon external entities to monitor resource availability across the network and feed such information to the infrastructure

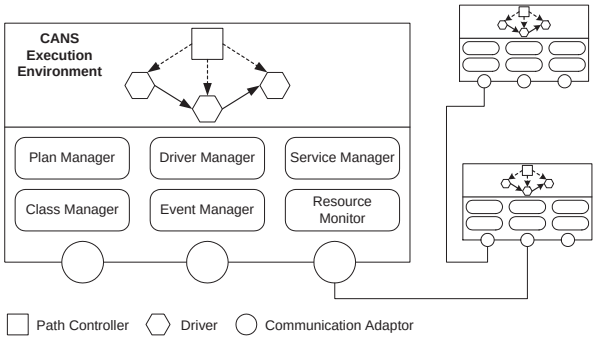


Figure 8.1: CANS Execution Environment

work is realized by a set of CANS-Enabled nodes, each runs an instance of the CANS EE. Augmented paths are deployed on these nodes.

Components (drivers) and Most parts of the CANS EE are written in Java; the current version runs on any Java-enabled node. The primary reason we choose Java for our prototype is because of its existing support for code mobility and controlled execution for downloaded code from different sites.

In this chapter, we describe in turn the overall structure of the CANS execution environment, several important interfaces, data communication and control interaction in the infrastructure, its support for legacy applications and components (referred to as services). We conclude this chapter by showing the whole procedure of setting up and reconfiguring a CANS path.

8.1 CANS Execution Environment

8.1.1 Overall Structure

The CANS Execution Environment (EE) serves as the node run-time system for components in augmented communication paths. The structure of the CANS EE is illustrated in Figure 8.1. It contains the following six major modules: *Plan Manager*, *Driver Manager*, *Service Manager*, *Event Manager*, *Resource Monitor*, and *Class Manager*. In this section, we provide a brief description of the functions provided by each of these modules.

The *Plan Manager* is responsible for constructing network-aware communication paths upon requests from applications. The planning algorithms described in Chapter 5 are used to calculate communication paths with optimized performance in accordance with application requirements and underlying network conditions. Moreover, this module is also responsible for partitioning the generated paths and sending path segments to all of the nodes involved in the path. The plan manager has both local and remote interfaces so that it is possible to run a plan manager on only a subset of the nodes along a path, a feature important for small devices that have very limited resources and computation capacity.

The *Driver Manager* is a support module for the plan manager. Its primary responsibility is to manage drivers within a CANS EE, i.e., to assemble, modify and remove driver graphs.

The *Service Manager* is used to control legacy components on local hosts, inside or outside of the CANS EE. Legacy components are called services in our framework, and distinguished from drivers in that they are not required to support the interface re-

quirements imposed on the latter. The service manager module provides APIs for creating, terminating, registering, and querying service instances. Besides, in order to process data using these services, the service manager connects active service instances with drivers in the execution environment.

The *Class Manager* is responsible for downloading code and instantiating driver instances as required. It is built as a custom class loader for the Java Virtual Machine supporting the EE. Upon receipt of a component graph, the *Class Manager* first tries to load all the class files of the corresponding components from its local repository. If the code is not available locally, the class loader downloads the class files from the location where it receives the graph. The use of the class loader mechanism gives us the flexibility of isolating the execution of components from different sites. Furthermore, by associating each class loader with a specific instance of the environment object, used by the drivers for interacting with the underlying EE, we can further customize the access of downloaded code from different sites.

The *Event Manager* is used to propagate events within an EE and across different EEs. Distributed events are the primary interaction mechanism in the CANS. Compared with a pre-wired implementation, the event paradigm (with simple publish and subscription primitives) is an effective, and much more flexible way for modeling complicated control logic among different entities in this component-based framework. Components can have different needs from the EEs, and are usually developed by many different providers. Additionally, the event model gives us the flexibility of extending the functionality built in the infrastructure. A more detailed description of the use of distributed events in CANS is deferred to Section 8.1.4.

The *Resource Monitor* is responsible for producing notifications of changes in re-

```
public class PathController
    implements PerformanceEventListener, ReconfigEventListener {

    private void init(AppPolicy policy, PathGraph graph);

    //process events
    public boolean processReconfigEvent(ReconfigEvent event);
    public boolean processPerformanceEvent(PerfEvent event);
    public boolean forwardEvent(Event event);

    //return address list of all nodes
    public synchronized List getRemoteAddressList();

    public String getPathId();

    //global/local reconfiguration
    public void lockForReconfiguration();

}
```

Figure 8.2: Path Controller Interface

source availability: once it detects a change in resource availability (for an individual path or for the whole network resource), the resource monitor generates a notification by raising a corresponding event. For changes occurring on the resource level, the current CANS implementation relies on external entities for monitoring resource availability and providing change information. Such information from the external monitor utility is transformed into events in CANS via the interface provided by the resource monitor.

8.1.2 Path Controller

To control augmented paths, an instance of a *path controller* is created in every EE for each path that is deployed to a set of CANS-enabled nodes. The most important

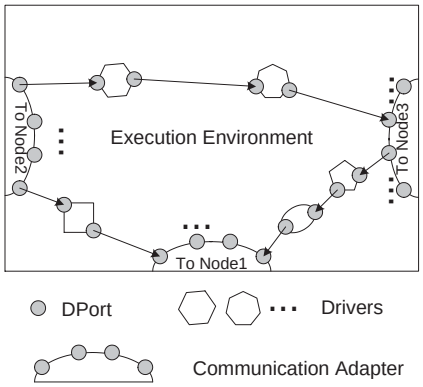


Figure 8.3: CANS Communication Adapter

function of the path controller is to monitor performance of the path and trigger reconfiguration whenever necessary. Most parts of the path reconfiguration protocol are implemented in the path controller, which controls components (and their data ports) to operate in accordance with the states in the reconfiguration process (as described in Chapter 6). For local reconfiguration situations, the path controller first needs to *lock* the region that it wants to modify. In addition to this, path controllers are also used to forward path-level events (events destined for all nodes along the path). The interface of the path controller is illustrated in Figure 8.2.

8.1.3 Communication Adapter

Data communication across networks in CANS is implemented with auxiliary CANS components called communication adapters. Communication adapters hide details of physical data transmission from drivers by exposing the same data port interface to drivers. Therefore, from a driver's perspective, sending data across the network is exactly the same as forwarding data to another driver in the same EE. CANS contains

several different types of communication adapters that transmit data between two EEs, or between an EE and an application that does not expose the data port interface.

Each communication adapter can support multiple input/output data ports to/from CANS paths. This allows multiple logical connections to be multiplexed on a single physical link (see Figure 8.3). A communication adapter can further exploit the transport mechanism that best matches the characteristics of the underlying network. Additionally, communication adapters can also encapsulate behaviors that permit them to adapt to and recover from minor variations in network characteristics. For instance, these adapters can be written to use one of several network alternatives, automatically transitioning between them to improve performance.

8.1.4 Event Propagation

Dynamic changes and control messages in the CANS infrastructure are realized as distributed events in the system. The use of the event paradigm gives us the flexibility in allowing different parts of CANS infrastructure to interact with each other, without requiring prior knowledge of their interfaces.

In CANS, any entity (including drivers) can raise arbitrary events as well as listen for specific ones. Event support is realized by a per-EE **Event Manager**, which is responsible for catching, firing, and transmitting events across the network. Event raising and firing is implemented using simple method calls and callback functions associated with the relevant components.

A CANS event contains a name, and the IDs of the event source and destination. Each of these values can be specified as a wildcard value. When the event manager receives an event (from the local EE or from the network), a template match is per-

formed between the event object and all registered templates. This approach is very flexible in that using this, different communication patterns (unicast, multicast, and subscription of multiple events) can be easily implemented.

There are two major types of CANS events: events from the local resource monitor, indicating a change in resource status, and events from components on the communication path. The first type of events is sent only to local components that have registered themselves as interested listeners. The second type of events, called path-level events, are first sent to the *path controller* (see Section 8.1.2), which is responsible for forwarding the event to the destination along the path. The path controller keeps track information such as driver location etc., and uses the event manager to transmit events across the network. For example, for events whose destination ID is the ID of a path, they are delivered to all nodes along the path; for events whose destination is (path ID, driver ID), they are sent to the node where the specified driver resides.

8.2 Interfaces of Components and Types

To facilitate dynamic composition, our framework relies on an interface to which drivers are required to adhere, and a type model (described in Chapter 4). To give some details on how all these concepts are implemented, in this Section we present the interfaces of components and types used in the CANS infrastructure.

```
public abstract class Driver implements Serializable{
    protected Driver(String id);
    public boolean init();
    //push/pull operations
    public abstract void push(DInPort input);
    public abstract void pull(DOutPort out);

    //lookup for ports
    public abstract DPort  getPort(String PortId);
    public List getAllPorts();
    //calculate types of output ports
    public abstract Map getOutputPortType
                        (Map inputPortTypes);

    //A incoming marker received
    public synchronized void incomingMarker(DPort srcPort,
                             int seqNo);
    //send out a marker
    public void outMarker(int seqNo);

    //event interaction
    protected void raiseEvent(Event event);
    protected void registerEventListener(String eventName,Object src,
                                         Object dest=null,
                                         EventListener listener);
    protected void removeEventListener(String eventName, Object src,
                                       EventListener listener);

    //for reconfiguration
    public boolean setStatus(int newStatus,String portId);

    public static void setExecutionEnvironment(DriverEE dee);
    public void reset();
}
```

Figure 8.4: Driver Interface

8.2.1 Interface of Components

The interface of drivers is shown in Figure 8.4. It contains four groups of methods, which respectively handle data port (DPorts) lookup, events, data transmission, and

path reconfiguration. Most of these methods are implemented in the base Driver class so that derived component classes can use them directly. Note that the Driver class has a static member of a driverEE object, an environment object that allows components to interact with the underlying EE. As we mentioned in Section 8.1.1, this field is initialized by the class loader to customize the access right of downloaded driver code.

The DPort interface is illustrated in Figure 8.5. It includes methods for setting up connections between different DPorts, and dispatching data between it and the owner driver. Moreover, a data port contains a status flag, which is controlled by the *path controller*. This flag is used to determine whether incoming data should be forwarded, buffered, or discarded according to the reconfiguration protocol discussed in Section 6.3.1. The DPort is an abstract class that contains implementations for common methods in this interface. Derived from the DPort class, there are two sub-classes: DOutPort and DInPort which support input or output operations respectively. Implementations of all of these classes are provided by the infrastructure.

8.2.2 Interface of Types

The interfaces for simple types and stream types are shown in Figure 8.6. The stream type is basically a stack of simple data types. The most important operations for them are 1) determining if two types are compatible, and 2) calculating the augmented part of a type instance in a type specific way when a type instance is passing through a network link, as described in Chapter 4.

```
public abstract class DPort implements Serializable {

    protected DPort(Driver owner, String portId);

    public  Driver getOwner();
    public  String getPortId();

    public  PortType  getPortType();
    public  void    setPortType(PortType type);

    public abstract void incomingMarker(int seqNo);
    public void setData(byte[] buffer);
    public byte[] getData();
    public boolean hasData();

    public void connect(DPort newLinkee)
                throws TypeIncompatibleException;
    public void disconnect();
    public DPort getLinkee();

    //status during reconfiguration
    public int getStatus();
    public void changeStatus(int status);

    public synchronized void reset();

    public final static int STATUS_ACTIVE=0;
    public final static int STATUS_BUFFER=0x1;
    public final static int STATUS_DISCARD=0x2;
}
```

Figure 8.5: DPort Interface

8.3 Support for Legacy Components or Applications

In this section, we describe how the CANS infrastructure supports legacy components (i.e. existing functionality encoded in a way that does not adhere to the required interface of drivers), and legacy applications that are CANS oblivious. For legacy components, we view them as if they were regular network services that provide content or

```
public abstract class DataType implements Serializable, Cloneable
{
    public abstract boolean isCompatible(DataType dt);
    public boolean equals(Object dt);

    //calculation of augmented part when passing a network link
    public LinkProperties passLink(LinkProperties link);

    public Object clone();
    //type rank
    public int getRank();
    public void setRank(int rankValue);

}

public class StreamType implements Serializable {
    public StreamType();

    public DataType getCurrentType(boolean peek);
    public void pushNewType(DataType newType);

    public Object clone();
    public boolean equals(Object obj);

    public LinkProperties passLink(LinkProperties link);
}
```

Figure 8.6: Type Interfaces

data processing functionality. CANS provides a general platform for integrating and controlling services (running inside or outside of the CANS EE); legacy applications are supported using an interception layer to bridge them to the CANS infrastructure.

8.3.1 Services

Services are modeled as providing content or data processing functionality. Unlike the constrained driver interface, services can export data using any standard protocol (e.g., TCP or HTTP), encapsulate heavyweight functions, process concurrent requests, and

maintain persistent state. Relaxing interface requirements permits use of legacy services; however, our framework does not support migration for services, requiring a service to manage its own state transfer. This design choice reflects the view that services are migrated infrequently and doing so requires protocols that are difficult to abstract cleanly.

CANS provides APIs to create, compose, and control services across the network. Services can run inside or outside the CANS EE. A service is required to register itself by providing a *delegate object* that can control the service and act on its behalf in interactions with the rest of the framework.

8.3.2 Support for Legacy Applications

The CANS infrastructure supports both CANS-aware and CANS-oblivious applications. The former just hook into the driver and service interfaces described earlier. For the latter cases, CANS provides an *interception layer* that is transparently inserted into the application and virtualizes its existing network bindings. The interception layer is injected using a technique known as API interception [31], supported on both Windows and Unix platform, using a variety of mechanisms, ranging from DLL import table modification to run-time rewrite of portion of the memory image of the application.

The general architecture of the interception layer is shown in Figure 8.7. The interception layer provides the application with an illusion of a TCP socket which can be bound to various interfaces (CANS or native network) for actual data transmission. An application specific policy responds to events (such as connect requests) delivered to it by the interception layer, which in turn influences the binding. Thus, enabling

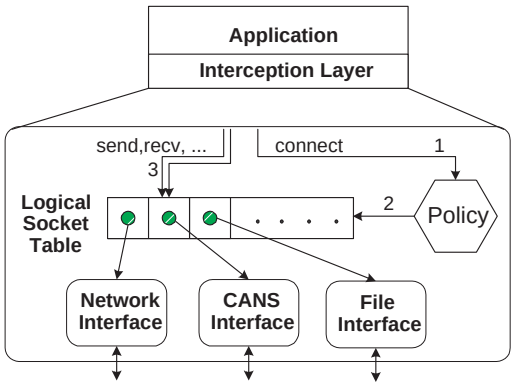


Figure 8.7: Architecture of the interception layer.

CANS support for a new legacy application would require only writing a specific policy for that application.

8.4 Procedures of Path Setup and Reconfiguration

To demonstrate how these parts of the CANS infrastructure work together, in this section, we briefly describe the lifetime of a typical CANS path.

To set up the path, the application needs to call the *Plan Manager* (on either the local or a remote host) directly or via the interception layer, with information about the server to access and its performance requirements. The next steps that follow are different depending on a centralized or distributed strategy is in use.

In the centralized case, the *Plan Manager* first determines a network route (and the resource availability) between the server and the client application (using a shortest path algorithm). With the selected route, the plan manager constructs the component graph and the mapping using the planning algorithms described in Chapter 5. It partitions the component graph, and sends these partitions to nodes along the path.

The distributed strategy works as follows: When the plan manager receives a request from the application, it routes the request towards the server. After the request arrives, the server (or the CANS node next to the server along the route) bounces back a planning request. This planning request is received by each of the nodes along the route, which calculates its portion of the communication path. In the cases where complementary planning is needed (see Chapter 5), it is calculated during the forward routing stage.

After the components graph is determined and communicated, every node along the path instantiates its components in the local EE. In addition, it also creates an instance of the *path controller* object for controlling this path.

The *path controller*, running on each node along a CANS path, monitors events that reflect the performance of the path. Whenever a path controller realizes that the performance does not meet the requirements, it triggers reconfiguration using the protocol described in Chapter 6.

When data transmission of a path completes, drivers and path controller of the path are removed from the EE, and any allocated resources released.

8.5 Summary

In this chapter, we have presented a prototype of our framework, the CANS infrastructure. We described the structure of the CANS EE, which, as the kernel of the infrastructure, provides complete support for realizing network-aware communication paths. In addition to drivers and CANS-aware applications, legacy components or services can be integrated into communication path using a delegate model; legacy

applications can also use the CANS infrastructure via an interception layer.

Chapter 9

Evaluation

To evaluate our framework, we have extensively experimented with the CANS infrastructure. In particular, our evaluation focuses on the following questions:

1. Can application specific functionality be automatically introduced into the network, and if so, what are the associated overheads?
2. Do our automatic path creation strategies bring applications considerable performance benefit?
3. Can desirable continuous adaptation behaviors be achieved using our automatic strategies for path creation and reconfiguration?
4. What are the fundamental benefits of path-based approaches as compared to end-point or proxy-based alternatives?

To answer these questions, we have carried out four studies.

In the first study, we measured the runtime overhead of the CANS infrastructure by examining its impact on bandwidth and latency of communication paths when trivial

forwarding drivers are in use. To investigate the overheads of component composition at a finer level of detail, the timeline of a more complicated augmented path was also examined.

In the second study, we investigated the performance advantages that our framework can bring to applications. In the experiments, we run a typical web access application under a wide range of network configurations, comparing achieved performance of the case where the CANS infrastructure was used and the situation where no adaptation support was provided.

In the third study, we examined the adaptation behaviors provided by our automatically constructed and reconfigurable communication paths. By running an image streaming application within a network environment where bandwidth changes frequently, we characterized how fast the CANS infrastructure could adapt to resource changes, and how good the automatically generated adaptation decisions were.

In the last study, we compared performance differences between a CANS-link path-based approach and other alternatives, i.e. end-point and proxy-based approaches. The real question underlying this study was to investigate whether the path-based approach is really necessary in terms of the benefits it delivers despite its complexity. The comparison was conducted by simulating the behavior of each of these approaches in the context of a large network topology. In addition to comparing the performance differences, the experiments also provide us insights into how exactly the constraints on adaptation locations affect both performance of individual paths as well as the whole network. The latter provdes an understanding, hitherto unavailable, of the network configuration under which one approach is preferred over others.

We start by discussing the experimental platform and applications, and then present the result and analysis of the four studies.

9.1 Experimental Platform

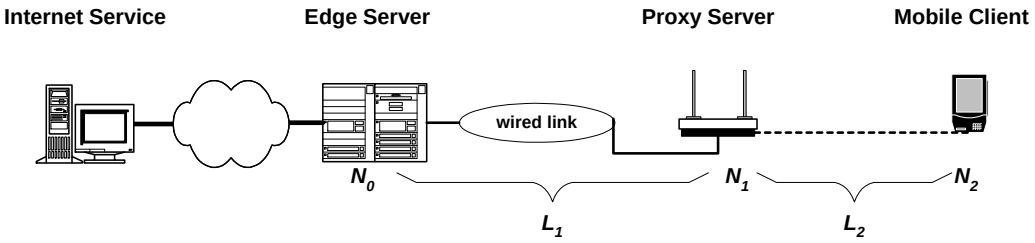


Figure 9.1: A typical network path between a mobile client and an internet services.

In most of our experiments, we consider a typical network path between a mobile client and an Internet server as shown in Figure 9.1. This platform models a mobile user using a portable device (N_2) such as a laptop or a pocket PC to access an Internet service in a shared wireless environment. The communication path from the device to the visited service typically spans (at least) three hops: a wireless link (L_2) connecting the user's device to an access point, a wired link (L_1) between the wireless access point and a gateway to the general Internet, and finally a WAN link between the gateway and the host running the service. We assume that CANS components can be deployed on three sites, the mobile device (N_2), a proxy server located close to the access point (N_1), or an edge server located near the gateway (N_0).[1]

In our experiments, bandwidth on links L_1 and L_2 can change dynamically. This

[1]Our use of the term "edge server" differs from its usage in content distribution networks. We use the term to refer to a host on the frontier of the network administrative domain.

either results from dynamic network traffic or users joining and leaving the shared wireless network N_1.

For our experiments, network configurations with different link bandwidths and computation capabilities are obtained by running CANS either on an appropriate selected actual hardware platform, or one emulated using "sandboxing" techniques that model a range of computation capacity and link characteristics by limiting CPU consumption of applications and the rate at which applications are allowed to send and receive messages [10].

The accuracy of the emulated behaviors using the "sandbox" techniques is discussed in Appendix B. The sandbox techniques give us the flexibility of controlling experiment parameters used in our experiments, making up for the absence of such control in current-day hardware. Additionally, the "sandbox" also provides resource availability information to CANS EEs.

9.1.1 Applications

We use two applications used throughout our experiments: a web access application and an image streaming application.

The web access application involves a browser client, which downloads web pages (both HTML page and images) from a standard web server. Our experiments used the Microsoft Internet Explorer(IE) browser. The communication path from IE is bridged into the CANS infrastructure using a CANS-aware HTTP Proxy.

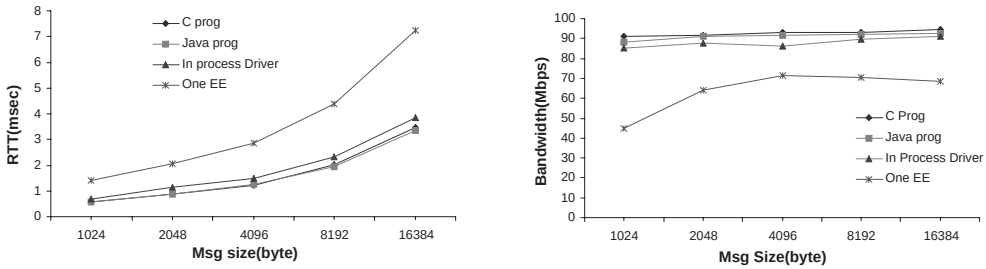
For this application, short response time is the major performance concern. Transcoding components can reduce download times under low-bandwidth network conditions

by dynamically compressing text and/or degrading image quality. Previous research has shown that such an approach is effective [18, 44]. In our experiments, we used this application to study whether an appropriately customized subset of these transcoding components can be automatically deployed to minimize download time for different network conditions.

The image streaming application is a Java Media Framework (JMF) application that continuously fetches and displays JPEG frames from an image server. To perform appropriately, this application requires that frames arrive at a certain throughput (i.e. frames per second), and once that is satisfied, prefers high quality data. The communication path between the client application and the image server can be augmented with two kinds of transcoding components capable of *filtering* and *resizing* images respectively. Each of these components can support multiple configurations: the `Filter` component can produce output images corresponding to different JPEG quality levels, while the `Resizer` component can generate output images with different scale factors. In our experiments, we used this application to investigate whether desirable adaptation behaviors can be achieved using our automatic path creation and reconfiguration strategies. This application also serves as the application for our simulation-based study that compares performance among end-point, proxy-based and path-based adaptation approaches in a large network setting.

9.2 Runtime System Overhead

To measure the runtime overhead of the CANS infrastructure, we run two experiments. First, we used a simple path containing only trivial forwarding drivers, and compared its performance (bandwidth and latency) with that of a direct TCP connection. In the second experiment, to investigate the detailed cost of component composition, we recorded the timeline for a more complicated CANS path and examined the actual overhead incurred in the interaction between different components in our implementation.



(a) Round Trip Time (b) Bandwidth

Figure 9.2: Latency and bandwidth impact of the CANS infrastructure.

9.2.1 Microbenchmarks

All measurements in this section were taken on a set of Pentium II 450Mhz, 128 MB nodes, running Windows 2000 and connected using 100 Mbps switched Ethernet.

Figure 9.2 shows the overheads introduced by CANS, measured in terms of how they impact communication performance between two communication parties. The applications used here are simple, standard applications to measure bandwidth and

round trip time. Each graph shows the round-trip time and bandwidth achievable for different message sizes for four configurations: C prog and Java prog refer to our baselines, corresponding to application and server programs that communicate directly using native sockets in C or Java respectively. In process Driver and One EE refer to basic CANS configurations; the former shows the case when trivial forwarding drivers (also called null drivers) and a communication adaptor are embedded into the application interception layer and indicates the basic overheads of driver composition, and the latter considers the case where the communication path includes null drivers on an intermediate host between the application and service.

Figure 9.2 shows that the In process Driver configuration introduces minimal additional overheads when compared with the Java prog configuration (less than 10% arising from extra synchronization and data copying), attesting to the efficiency of our driver design and composition mechanism. On the other hand, the One EE configuration does show marked degradation in performance, primarily because of context switch costs and the fact that the transmitted data has to traverse across application-level and network-level in the communication protocol stack four times instead of two times. However, given that intermediate EEs are intended to be used across different network domains in the Internet where other factors dominate latency and bandwidth, such overheads (an extra cost of 1-2 milliseconds or restricting achieved bandwidth to be around 70 Mbps bandwidth) is unlikely to have much overall impact.

9.2.2 Timeline of an augmented path

To investigate the detailed runtime overhead caused by component composition, we recorded the timeline of a more complicated path. The path we recorded was gen-

erated for the web access application. The path structure is illustrated in Figure 9.3. It contains components that can reduce sizes of images and HTML pages; the path branch for images contains components that can degrade image quality (`ImageFilter`) and decrease the image dimensions (`ImageResizer`); the branch for HTML pages contains compression/decompression components using the `Zip` algorithm. Details of these components or the reason for such a configuration are deferred to the next Section. The focus of this section is on the runtime overhead incurred by interaction between adjacent components along the path, i.e. the cost of composition.

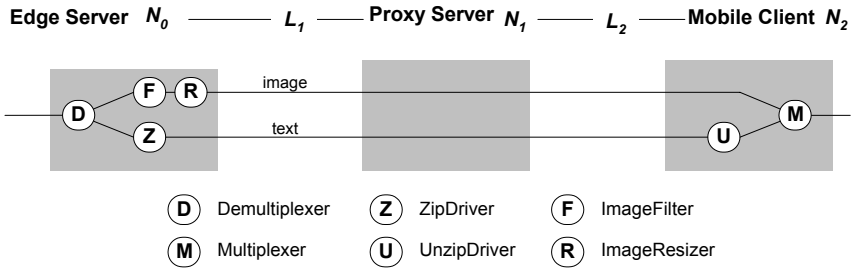


Figure 9.3: An augmented path for the web access application.

Figure 9.4 shows the overall timeline for the path when a HTML page and six embedded JPEG images are downloaded. This timeline is broken down into individual operations performed by the CANS execution environment and the components themselves for processing a single text and image packet. The original client request results in the downloading of the text portion of the page, and is followed by requests for each of the six contained images. A request is sent to the web server through N_1 and N_0. Text responses comprise several packets, each of which passes through the Demultiplexer and Zip drivers on the edge server, and the Unzip and Multiplexer drivers on the client before being delivered to the browser application. Similarly a

response to an image request comprise multiple packets, each of which flow through the Demultiplexer, ImageFilter, and ImageResizer drivers on the edge server and the Multiplexer on the client before being delivered to the application.

The timeline shows that for this particular application, CANS overheads are negligible because the dominant contributor to response time is actually the round-trip between the edge server and the central server (0.2 seconds on the text path and 0.16 seconds on the image path). Even if this were not the case, CANS run-time overheads (shown hatched in the figure) for retrieving data from the network and supplying it to each driver in turn are small for all but very fine-grained components (the Demultiplexer and Multiplexer). For the components used in this experiment, CANS incurs an average cost of about 25μs for each driver invocation, reflecting the cost of several method calls between adjacent components, which is acceptable for most applications.

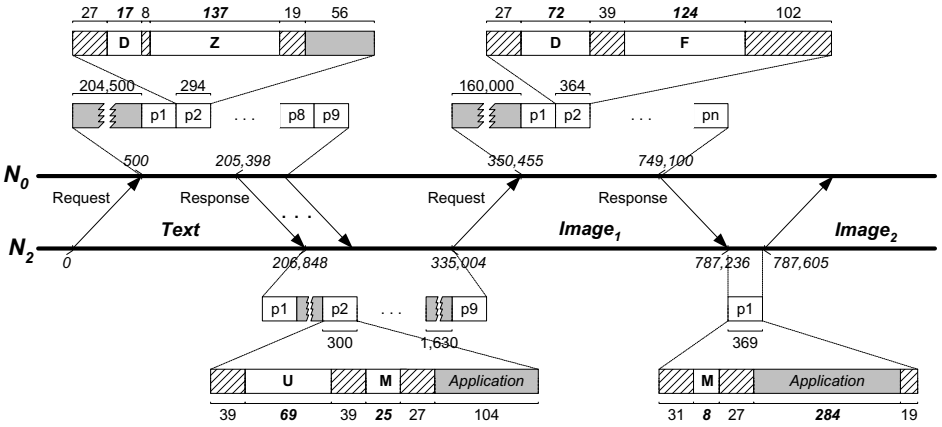


Figure 9.4: Timeline of requests and responses (all times are microseconds). The blocks marked **D**, **M**, **Z**, **U**, and **F** correspond to the executions of the respective components. Communication overheads, including wait times, are shown using gray, whereas CANS overheads are shown using hatched blocks. *Application* refers to the overhead of communicating the data to the client application.

9.3 Effectiveness of Automatic Path Creation

To study the performance advantages obtained by using automatically generated paths, we experimented with the web access application under a wide range of network configurations and compared the performance of CANS paths with that of direct TCP connections.

Components used with the web access applications include: single-configuration *ImageFilter* and *ImageResizer* components, which can degrade image quality to a factor of 0.2 and reduce image size to a factor of 0.2 respectively, and *Zip* and *Unzip* components, which work together to compress text pages. The load and bandwidth factor values were obtained by profiling component execution on representative data inputs: a web page containing 14 KB text and six 25 KB JPEG images (see Appendix A for the profile information). In this experiment we used the same data inputs that the components were profiled on. This is a simplifying assumption, but reasonable given our primary focus here was evaluating whether our approach could effectively construct the "best" communication path for different network conditions. Evaluating the effectiveness of the approach when component characteristics may be imprecise is examined in our next set of experiments.

To model different network conditions likely to be encountered along a mobile access path, we defined twelve different configurations listed in Table 9.1. These configurations represent the network bandwidth and node capacity available to a single client, and reflect different loading of shared resources and different mobile connectivity options.[2] These configurations are grouped into three categories, based on

[2]The bandwidth between the internet server and edge server available to a single client is assumed to be

whether the mobile link L_2 exhibits cellular, infrared, or wireless LAN-like characteristics. Five of the configurations correspond to real hardware setups (tagged with a *), the remainder were emulated using "sandboxing" techniques that constrain CPU, memory, and network resources available to an application [10]. The computation power of different nodes is normalized to a 1 GHz Pentium III node with 256M bytes 800MHz RDRAM.

Table 9.1 also identifies, for each platform configuration, the plan automatically generated by CANS for the web access application. The plans themselves are shown in Figure 9.5. To take an example, consider platform configuration 7 for which the path creation strategy generates Plan C. The reason for this plan is as follows. Since link L_1 has high bandwidth while L_2 has moderate bandwidth, there is a need to reduce image transmission size, which is accomplished using the *ImageFilter* component. The *Zip* and *Unzip* drivers help improve download speeds by trading off computation for network bandwidth. Both the *ImageFilter* and *Zip* components are placed on the proxy server, because it has more capacity than the edge server. While this explanation justifies the generated plan, we note that the plans themselves were mechanically generated use the algorithms described in Chapter 5.

Figure 9.6 shows the performance advantages of the automatically generated plans when compared to the response times incurred for direct interaction between the browser client and the server (denoted Direct in the figure). The bars in Figure 9.6 are normalized with respect to the best response time achieved on each platform (so lower is better). In all twelve configurations, the generated plans improve the response time metric, by up to a factor of seven. Note that the lower response times come at the

10Mbps.

Platform	*Edge Server* (N_0)	L_1	*Proxy Server* (N_1)	L_2(bps)	*Client* (N_2)	*Plan*
1	Medium	Ethernet	High	19.2 K	Cell Phone	A
2	Medium	Ethernet	High	19.2 K	Pocket PC	A
3*	High	Fast Ethernet	Medium	57.6 K	Laptop	B
4*	High	Fast Ethernet	Medium	115.2 K	Laptop	B
5	Medium	Ethernet	High	384 K	Pocket PC	A
6*	High	Fast Ethernet	Medium	576 K	Laptop	B
7*	Medium	Fast Ethernet	High	1 M	Laptop	C
8	Medium	Ethernet	High	3.84 M	Pocket PC	D
9	Medium	Ethernet	High	3.84 M	Laptop	D
10	Medium	DSL	High	3.84 M	Laptop	B
11	Medium	DSL	Low	3.84 M	Laptop	B
12*	Medium	Fast Ethernet	High	5.5 M	Laptop	E

Relative computation power of different node types
(Normalized to a 1 GHz Pentium III node with 128 MByte 800MHZ RDRAM):
High = **1.0**, Medium = **0.5**, Laptop = **0.5**, Low = **0.25**, Pocket PC = **0.1**, Cell Phone = **0.05**
Link bandwidths:
Fast Ethernet = **100 Mbps**, Ethernet = **10 Mbps**, DSL = **384 Kbps**

Table 9.1: Twelve configurations representing different loads and mobile network connectivity scenarios, identifying the CANS plan automatically generated in each case.

cost of degraded image quality, but this is to be expected. The point here is that our approach *automates* the decisions of when such degradation is necessary.

Figure 9.6 also shows that different platforms require a different "optimal" plan, stressing the importance of automating the component selection and mapping procedure. In each case, the CANS-generated plan is the one that yields the best performance, also improving performance by up to a factor of seven over the worst-performing transcoding path.

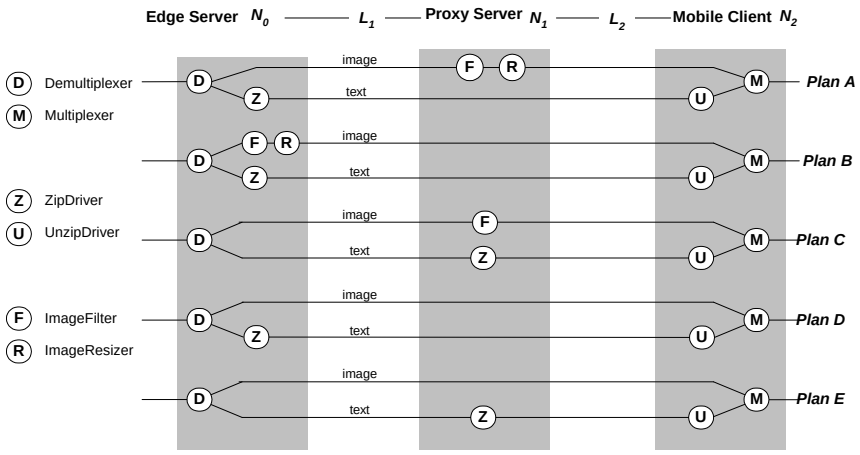


Figure 9.5: Component placement for the five automatically generated plans.

9.4 Dynamic Adaptation Behaviors

To study the adaptation behavior achieved using the CANS infrastructure, we experimented with the image streaming application in a dynamic network environment. The experiment modeled the following scenario: initially a user receives a bandwidth allocation of 150 KBps on the wireless link (L_2), which then goes down to 10 KBps in increments of 10 KBps every 40 seconds (modeling new user arrivals or movement away from the access point) before rising back to 150 KBps at the same rate (modeling user departures or movement towards the access point). The communication path is allocated a (fixed) computation capacity of 1.0 (normalized to a 1 GHz Pentium III node) on nodes N_1 and N_2 respectively and a bandwidth of 500 KBps on L_1. The rationale for these choices is that N_1, N_2, and L_1 are wired resources and consequently more capable of maintaining a certain minimum allocation (e.g., by employing additional geographically distributed resources) than the wireless link L_2.

In this experiment, we started with the base mechanisms (base planning algorithm

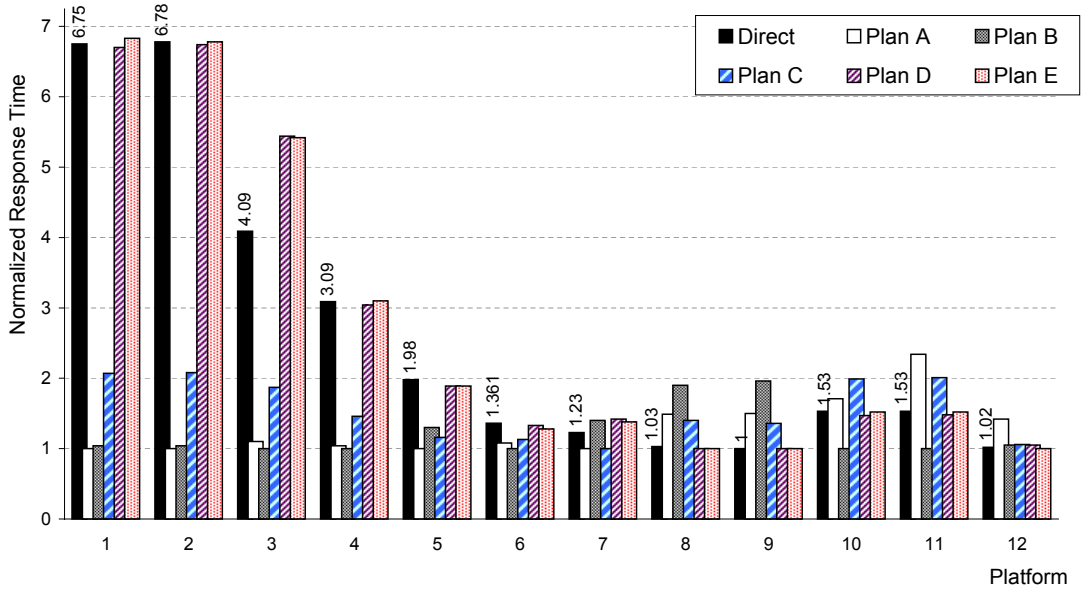
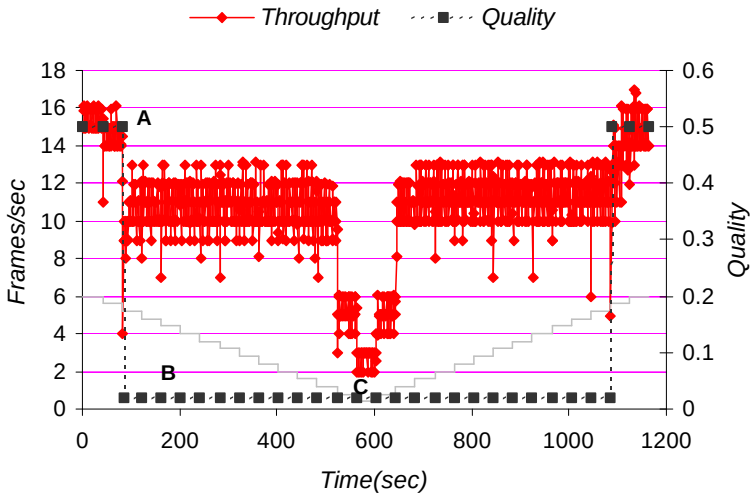


Figure 9.6: Response times achieved by different plans for each of the twelve platform configurations compared to that achieved by direct interaction. All times are normalized to the best performing plan for each configuration.

+ global reconfiguration), and show the incremental benefits on adaptation behavior from each of the schemes described in Chapter 5 and 6.

9.4.1 Base Mechanisms

In the first step, the components used with the image streaming example included the `ImageFilter` and `ImageResizer` used in section 9.3 (which degrade image quality or reduce image size by a factor of 0.2). As mentioned earlier, the application requirement is for the throughput to be in the range of 8 to 15 frames per sec. Within that range, better image quality is preferred. We started with the base planning strategy described in Section 5.4. Since the strategy can only optimize one of these metrics at a time, we chose to optimize throughput. The component parameters were



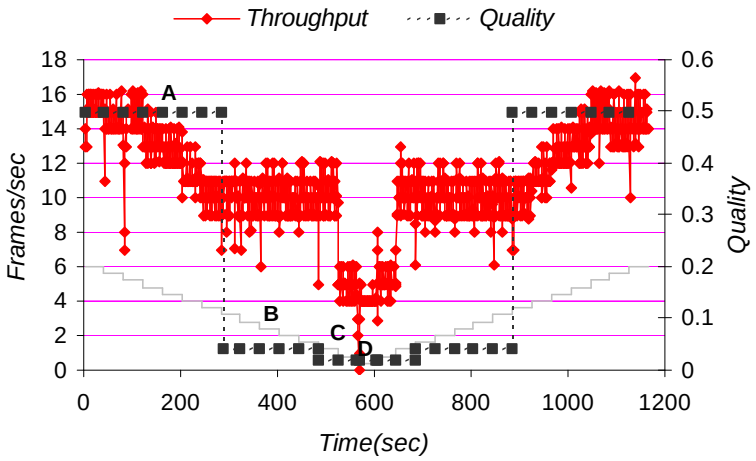
Plan	N_0	N_1
A	**F**	–
B	**F**	**R**
C	**F R**	-

F: **F**ilter, **R**: **R**esizer

Figure 9.7: Performance with the Base Planning Algorithm

obtained by profiling their behavior on a 25 KB JPEG image (quality assumed to be 1.0), one of a set of images ranging in size from 20–30KB repeatedly transmitted by the server. The profiled values of computation load and bandwidth impact factor for various components are listed in Appendix A.

Figure 9.7 shows the throughput and image quality achieved by the communication path over the 20 minute run of the experiment; the plans automatically deployed by CANS are shown in the right table. The plot needs some explanation. The light-gray staircase pattern near the bottom of the graph shows the bandwidth of link L_2 normalized to the throughput of a 25 KB image transmitted over the link; so, a link bandwidth of 150 KBps corresponds to a throughput of 6 frames/sec, and a bandwidth of 10 KBps corresponds to a throughput of 0.4 frames/sec. The dashed black line corresponds to the quality achieved by the path. The jagged curve shows the number of frames received every second; because of border effects (a frame may arrive just after the measurement), this number fluctuates around the mean. The plateaus in the



Plan	N_0	N_1
A	**F**	-
B	**R**	-
C	**F**	**R**
D	**F R**	-

F: **F**ilter, **R**: **R**esizer

Figure 9.8: Performance with Range Planning

quality curve are labelled with the plan that is deployed during the corresponding time interval.

The results in Figure 9.7 show that the plans automatically created and dynamically deployed by CANS do improve application throughput over what a static configuration would have been able to achieve. However, it also points out several deficiencies:

- Always trying to maximizing the throughput may sacrifice image quality unnecessarily, failing to meet application performance preference.
- The reconfiguration at 80 seconds from Plan A to Plan B is seemingly unexplainable given that it was initiated to improve application throughput, not to reduce it. A closer examination identified this problem to be caused by the fact that component behavior for the `ImageResizer` component did not match profiled behavior when the input was a filtered image as opposed to the original. A similar problem exists for Plan C.

9.4.2 Range Planning

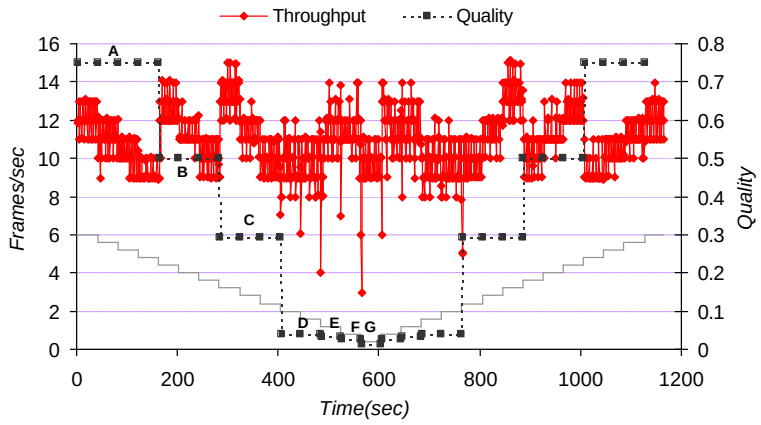
To address the first problem, we applied the range planning algorithm (Section 5.5) to this application, and obtained the result shown in Figure 9.8. Comparing with Figure 9.7, we can see two improvements. First, the range planning system retains Plan A for much longer than before (till 280 seconds into the experiment), choosing not to reconfigure while the throughput is still within the desired range. Second, the system employs an additional plan that falls between Plan A and B chosen in Figure 9.7 and represents a tradeoff that compromises on achieved throughput (while still ensuring that it is within the desired range) to improve quality. Such gradual decrease/increase in image quality is desirable adaptation behavior expected by end users.

9.4.3 Component Model

To address undesirable adaptation caused by inaccurate component parameters, we incorporated two improvements.

First, we allowed both components in our image streaming example to take on multiple configurations: nine `Filter` configurations corresponding to quality values 0.1 to 0.9, and eight `Resizer` configurations corresponding to scale factors of 0.1 to 0.8.

Second, we exploited the *class profiling* described in section 5.1.2. We profiled the components with three types of image quality: high(1.0), medium (0.5) and low (0.1). The parameters (comp, bwf) of these components used in path calculation are determined by the incoming image quality.



Plan	N_0	N_1
A	**F**(4)	-
B	**F**(3)	-
C	**F**(1)	-
D	-	**R**(1)
E	**R**(1)	**F**(5)
F	**R**(1)	**F**(3)
G	**R**(1)	**F**(1)

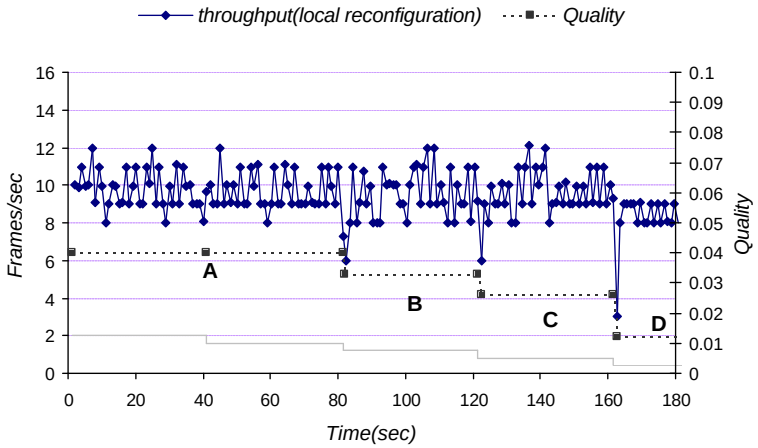
F: **F**ilter, **R**: **R**esizer

Figure 9.9: Performance with Multi-Configuration Components and Class Profiling

Figure 9.9 shows the resulting performance and associated plans. There are three obvious improvements over Figure 9.8. First, the throughput is kept in the required range for the whole duration of the experiment (except for transition points caused by reconfigurations). Second, the image quality changes more smoothly than what was previously shown in Figure 9.8. Instead of 3 configurations (quality levels), there are 7 different plans, permitting smoother variations in path quality. Finally, the low costs of switching between two configurations of the same component is reflected in transitions from plans A to B, and B to C, which hardly disrupt the achieved throughput unlike the associated cost for introducing a new component (transition between plan C and D).

9.4.4 Reconfiguration Overhead and Benefits of Local Reconfiguration

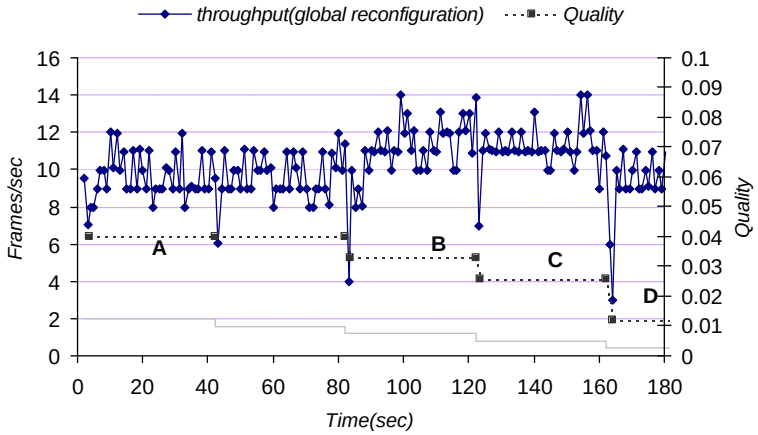
Reconfiguration may introduce interruptions in data transmission, therefore a short reconfiguration time is important for providing better user experience. To investigate



Plan	N_0	N_1
A	-	**R**(1)
B	-	**R**(1),**F**(5)
C	-	**R**(1),**F**(3)
D	-	**R**(1),**F**(1)

F: **F**ilter, **R**: **R**esizer

Figure 9.10: Performance of Local Reconfiguration



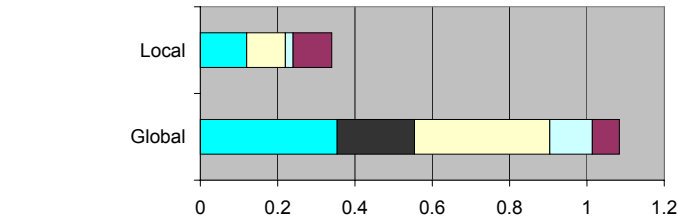
Plan	N_0	N_1
A	-	**R**(1)
B	**R**(1)	**F**(5)
C	**R**(1)	**F**(3)
D	**R**(1)	**F**(1)

F: **F**ilter, **R**: **R**esizer

Figure 9.11: Performance of Global Reconfiguration

the cost of data path reconfiguration in CANS, we used the image streaming application, and measured the reconfiguration overhead of local and global reconfiguration. In both cases, we measured the cost for level 3 reconfiguration, i.e. the reconfiguration that provides exactly-once and in-order delivery semantics for data transmission.

To emphasize the difference in behaviors between local and global reconfiguration, we closely examined the portion of the experiment between 400 seconds and 600



	Global	Local
5. Firstst Message	0.07	0.1
4. Reconfiguration	0.11	0.02
3. Flush-Data	0.35	0.1
2. Partition-Shipping	0.2	0
1. Planning	0.354	0.12

Figure 9.12: Reconfiguration Cost

seconds, corresponding to a bandwidth range of 50 KBps to 10 KBps. Unlike global reconfiguration which partitions the `ImageResizer` and `ImageFilter` portions of the data paths in plans B, C, and D, so that they run on both nodes N_0 and N_1 to obtain a slightly higher value of throughput, local reconfiguration chooses to both calculate the plan and deploy the components on the same node, thereby avoiding the cost of coordination across nodes. The cost however is that the local reconfiguration does not quite achieve the same throughput as the global case, achieving 10 frames/sec instead of 12. Note that this is still within the desired range, otherwise global reconfiguration would have been triggered.

A breakdown of the reconfiguration costs for the bandwidth change event at 480 seconds in the two cases is shown in Figure 9.12. The total reconfiguration time is 1.08 seconds and 0.35 seconds for the global and local case respectively. To map these overheads to the 6-stage reconfiguration process described in section 6.3.1: *Partition-*

Shipping is the overhead for delivering new plans to nodes (stage 1); *Flush-Data* covers stages 2 and 3; *Reconfiguration* is the time for stages 4 and 5; and *1st Message* is the time for delivering the first segment to the downstream node, i.e. stage 6.

This figure shows that the major contributors to shorter reconfiguration times in the local mechanism are the first 3 stages of reconfiguration: shorter planning time, which is the result of shorter network paths; and shorter overheads for partitioning the plan, flushing data belonging to the old plan, and deploying the new plan, all of which benefit from the fact that all required coordination occurs locally and there is less data in transit. It should be noted that during the first 3 stages (including planning) of reconfiguration, data keeps flowing downstream. So the suspension period of data transmission is about 0.18 seconds for the global case and 0.12 seconds for the local case.

Note that the time of the first three stages (including planning) basically reflects the inertia of data paths, in which the existing path is still in use after a resource change is detected. The difference (about 0.68 seconds) between global and local mechanisms means that using the local mechanism can substantially increase the responsiveness of the data path. This observation is hold out by Figures 9.10 and 9.11, which show throughput for local and global reconfiguration mechanisms respectively. From these figures, we can observe that the use of local reconfigurations does result in more stable throughput during reconfiguration (look especially at the first reconfiguration that happens at the 80 second point in the figure, which corresponds to the 480 second point in the original experiment).

9.5 Overall Benefits of Path-Based Approaches

To compare the performance of our approach with that of other alternatives, i.e., end-point and proxy-based approaches, we carried out a study that characterize the performance that can be achieved using each of these approaches. In particular, our goal was to investigate the following questions: What is the performance impact of placing constraints on adaptation location? Under which network conditions is one kind of approach preferred over the others? Is the additional complexity of the path-based approach, which requires distributed control over the network, really necessary?

We investigated these questions by simulating the behaviors of different approaches in the context of a large-scale network. We compared the performance of these approaches for different network configurations, load levels, and server/clients properties. In our simulation, each of these adaptation approaches tries its best to sustain as many connections as possible with performance of individual paths optimized as much as possible.

In this experiment, we used the strategies described in Chapter 5 through 7 for creating and reconfiguring paths, and managing resources. These strategies, though designed for our path-based infrastructure, are approach-neutral in that they do not introduce bias for any of these approaches, which differ only in the constraints on adaptation location. These strategies can be uniformly applied to the end-point, the proxy-based and our path-based approaches without affecting the fairness of the conclusions drawn from our study.

9.5.1 Methodology and Simulation Scenario

In order to study the performance of those adaptation approaches under different network conditions, we adopt a simulation-based methodology. Using a detailed simulator modeling a typical large-scale network where multiple concurrently-active clients download media content from server sites, we characterize the performance of the three approaches — end-point, proxy-based, and path-based. We provide an overview of our simulation scenario and performance metrics of interest below, deferring a detailed description of the specific parameters to the next section.

Simulated Network. The network modeled in our simulation is depicted in Figure 9.13. The network contains multiple ISP regions, each of which is modeled as a centralized gateway/proxy node providing a connection to the Internet backbone. The server and client nodes in the network are attached to one of these ISP nodes using various connectivity options.

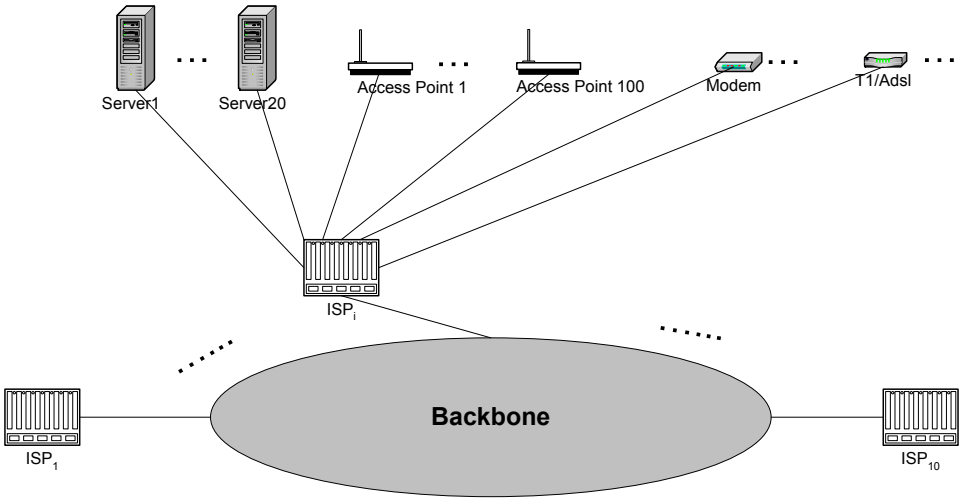


Figure 9.13: Experiment Network Topology

Application Behavior. The simulation models users connecting to server nodes from client nodes to download and display streaming media content. The connection is released once the download session is completed (which can happen either after the content is completely downloaded, or when the download task is cancelled by the user). To display the received content appropriately, the throughput of a download path is required to be in some specific range (i.e., a certain frame rate). When the available bandwidth is insufficient to meet the requirement, several components can be used to reduce bandwidth consumption.

9.5.2 Simulation Settings

Application Performance Requirements In our simulation, every client downloads continuous JPEG image frames (with an average size of 4K bytes) from a server site. In order to display the received content appropriately, the throughput of a download path is required to be in the range of 10 to 16 frames per seconds; within this range higher data quality is preferred.

Possible components that can be used with these paths include an *image-filter* and an *image-resizer*, which reduce bandwidth consumption by degrading image quality or reducing image size respectively. As in Section9.4.3, these components support 9 and 8 different configurations respectively; in each case the n^{th} configuration reducing image quality or size by a factor of $n/10$. Details about the `load` and `bwf` values of each operator are shown in Appendix A.

Network Characteristics The topology of our simulated network was shown earlier in Figure 9.13. For the results reported here, the network is assumed to comprise ten

ISPs. Each ISP is connected to the Internet backbone via an OC48 (2.488Gbps) link and includes 20 media servers, 100 public IEEE802.11b (6.0Mbps[3]) access points, and a number of client sites.

Connectivity options for clients include T3 (44.73Mbps), T1 (1.544Mbps), ADSL (1.5Mbps), Dialup (56Kbps), and IEEE802.11b connections (via the public access points). The T3, T1 and ADSL links are assumed to have sufficient bandwidth for the media application, while Dialup connections are incapable of meeting throughput requirements without the use of compression components. For wireless connections, available bandwidth is dependent on the load of the access point and may sometimes necessitate compression components along the path.

At each ISP, we model the arrival of clients as a Poisson process; the arrival rate of clients is a parameter that can be adjusted for different load levels. Once initiated, the duration of a download session is assumed exponentially distributed with an average of 1 minute.

Media servers within each ISP fall into one of two configurations. One fourth of the servers are categorized as *large sites*, with high-bandwidth connections to the ISP node (via an OC12 link operating at 622Mbps) and a computation budget uniformly distributed between 100 to 200 units.[4] The remainder three fourth of the servers are categorized as *small sites*, with relatively lower-bandwidth connections to the ISP node (an OC3 link operating at 155Mbps) and a smaller computation budget uniformly distributed between 10 and 100 units.

[3]We assume a 55% bandwidth utilization of an IEEE802.11b network.

[4]One unit is normalized as a computer with a Pentium III 1GHZ processor and 256MByte 800MHz RDRAM.

Adaptation Approaches Our simulation considered five different adaptation approaches: the end-point approach, the proxy approach, an approach that uses servers in addition to proxies (labeled as *server+proxy*), the path-based approach, and a path-based approach without reconfiguration support (labeled as *path-reconfig*). The last approach clarifies the benefits of dynamic adaptation; communication paths in this approach can adapt to different network conditions only at path-creation time. As mentioned earlier, the first four approaches represent different constraints on where adaptation is allowed. For the *end-point* approach, only the server node and the client node of a communication path can be involved in adaptation. The *proxy* approach is allowed to use client nodes and client-side ISP nodes. The *server+proxy* approach represents an intermediate point, which, in addition to nodes used by the proxy approach, can also use server nodes for adaptation. Finally, the *path* approach can use all four nodes along a communication path: the server node, the server-side ISP node, the client-side ISP node, and the client node.

To make a fair comparison between these approaches, our studies used the same total computation resource budget in each case.[5] In the end-point approach, all resources reside on server sites. For the proxy approach, all resources on server sites are aggregated on the ISP nodes they attach to. For the server+proxy approach and the path approach, a portion of the computation budget of every server site is moved to its ISP node using the strategy described in Section 7.2. The redistribution assumes that requests from clients are uniformly distributed among all server sites. Our study

[5]The computation budget refers only to resources available for path transcoding and compression operations. Sufficient resources are assumed available on the server and proxy nodes for data retrieval from disk and forwarding through the protocol stacks.

also examines situations where this assumption does not hold, providing insights into how performance is affected by inaccuracies in client traffic models.

Performance Metrics. Our simulations characterize two major performance metrics. The first is the aggregate time of all paths when the throughput of the path is in the desired range. We refer to this as the `InRange` time, i.e., the time where paths stay in the `InRange` state of the state diagram shown in Figure 7.1. Another possibility for this is the aggregate `InRange` time weighted by data quality of the communication path. Because we have observed the same behavior between the "weighted" and "original" `InRange` time in our experiments, we report only on results for the original `InRange` time.

The second performance metric is the total number of connection failures due to insufficient resources. Connection failures result from admission control, which actively rejects any incoming connection request if the initial planning cannot produce a communication path that meets the performance requirements.

In addition to the aggregate performance data for the whole network, we also collected data for different types of servers and clients to further examine how different adaptation approaches perform towards different types of servers or clients. In particular, we report on data of server sites that have the maximum or minimal computation budget (i.e. computation resources before redistribution for the path-based approach), and of clients that use different connectivity options.

Reconfiguration Overheads Path reconfiguration overheads in our study are modeled after the reconfiguration process described in Section 6.3.1. Specifically, it contains

the following six parts:

- Detection of changes in resource availability. In our simulation, network resources themselves are responsible for allocating partitions for individual paths (using the strategy described in Section 7.1.1). Therefore, the delay of detecting a change of resource availability is basically the time for delivering notifications. Since a notification is a small message that can be embedded in the regular data stream,[6] we model the delivery time as the total network link latency between the resource and the receiver.

- Planning. In general, the time for calculating a new path is highly dependent on the planning algorithm, but can be significantly reduced by employing a path cache of previously generated solutions. Given that attributes of most paths in our study (content type, client connectivity, resource availability) are likely to be clustered in a small range, we expect a high hit rate from such a cache. Consequently, we assume that planning incurs negligible overhead, modeling the situation where new plans are almost always obtained directly from the cache.

- Distribution of the new plan. New plan partitions need to be distributed to every node, participating in the reconfiguration, along the communication path. This is done by sending, in parallel to all these nodes, a data packet containing the plan partition of the receiving node. The packet itself has a size that is plan-dependent, and incurs latency dictated by the available bandwidth allocated to the path.

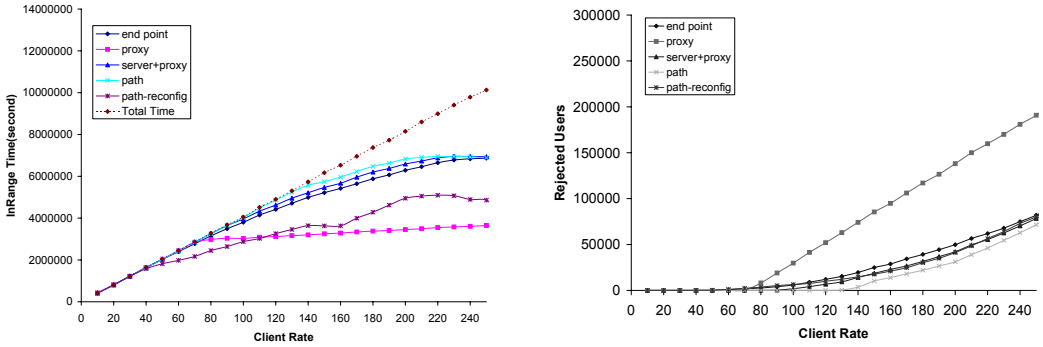
- Flushing data in transmission. The protocol ensures semantic continuity of data

[6] For example, the outbound data mechanism in TCP can be used for delivering such notifications.

transmission by flushing any incomplete data segments in transmission or internal state built up in operators (see Section 6.3). We model the overhead of this step in the simulation as the time required for transmitting the required segments.

- Deployment of new operators. Because operators are reusable and contain only soft state, the time for replacing old components with new operators on a node is usually a constant. In our study we use a value of 100 milliseconds, which is consistent with that observed in our previous experimentation with the CANS infrastructure (Section 9.4.4).
- Resumption of data transmission. The final step resumes data transmission through the new path. In the simulation, this step is assumed to incur negligible overhead.

In the rest of this section, we first report on the performance achieved by different adaptation approaches with client traffic uniformly distributed among the various server sites for a particular client connectivity profile. We then separately examine how performance is affected by non-uniform traffic distribution (where "hotspot" servers receive a larger share of the connection requests), and when the client connectivity profile is changed (with different fractions of clients using high-bandwidth and low-bandwidth links). In each case, we simulate the network for 4 minutes, recording data only for sessions that are started within the last 2 minutes, i.e. after the network reaches a stable state (recall the average length of a download session is 1 minute). The measurement ends at the 4 minute mark.



(a) Aggregate InRange Time (b) Aggregate Connection Failures

Figure 9.14: Aggregate Performance under Uniform Load Distribution.

9.5.3 Performance under Uniform Load Distribution

This configuration uniformly distributes client requests among all server sites, varying client arrival rates at each ISP from 10 to 250 clients per second. These rates correspond to 6000 to 150,000 active paths simultaneously existing in the network. The client connectivity profile is fixed as follows: 25% use links with sufficient bandwidth (T1/T3/ADSL), 25% use Dialup, and the remaining 50% use wireless connections. We examine the impact of changes from this profile later in Section 9.5.5.

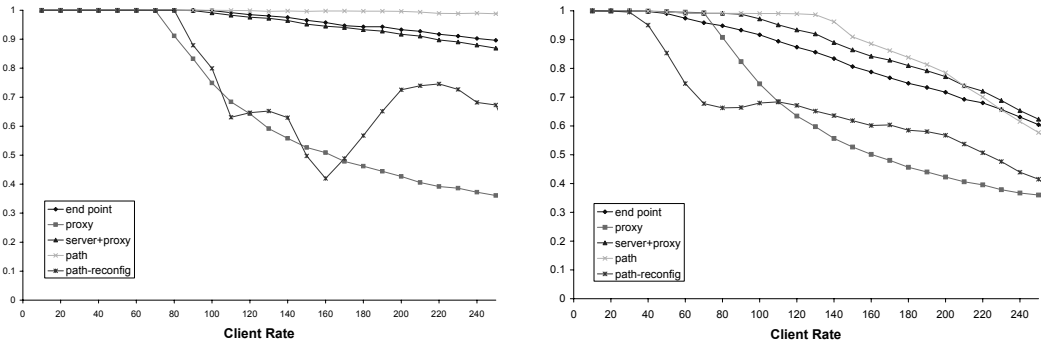
Analysis of Aggregate Performance

The aggregate performance achieved by different adaptation approaches for this configuration is shown in Figure 9.14. From Figures 9.14(a) and (b), it can be observed that all four adaptation approaches that include reconfiguration support perform very well when the network is lightly loaded. However, after the load increases to some level (client rate=80 in Figure 9.14(a)), the performance of the proxy approach is the first to reach saturation. This is explainable by the following: since adaptation can only occur on the node before the last hop, all paths end up consuming considerable

bandwidth in the network core, consequently saturating this portion of the network much faster than other approaches. Once the network gets saturated, further increases in InRange time are still possible, albeit at a much smaller rate, because of local loops (a client downloads contents from a server that is attached to the same ISP).

Compared with the proxy approach, the end-point approach performs better (with higher InRange Time and fewer connection failures), especially after the "saturation" point of the proxy approach. This is expected because the end-point approach uses server sites to do image filtering or resizing, and does not waste bandwidth on the network links. However, it can also be observed from the Figure 9.14(b) that the end-point approach starts to reject connections early, even when the network is lightly loaded. These rejections mainly come from clients that use weaker links such as Dialup to access small sites with limited computation capacity.

Figures 9.14(a) and (b) also show that the path-based approach provides the best performance at all load levels. The InRange time of the path-based approach is up to 12% and 97% higher than that of the end-point approach and the proxy approach respectively. The number of connection failures of the path-based approach is also much lower. For example, for a client rate of 200 connections/second, the end-point approach rejects 59% more connections and the proxy approach rejects about 343% more connections than the path approach. The reason for this behavior is because the path-based approach combines the advantages of both proxy and end-point approaches. On one hand, similar to the end-point approach, the path-based approach can utilize upstream nodes along a communication path to ensure that network bandwidth is not wasted; and on the other, similar to the proxy approach, the path-based approach can set up shared resource pools across the network, permitting overloaded



(a) Normalized InRange Time for T3/T1/ADSL Clients

(b) Normalized InRange Time for Dialup/Wireless Clients

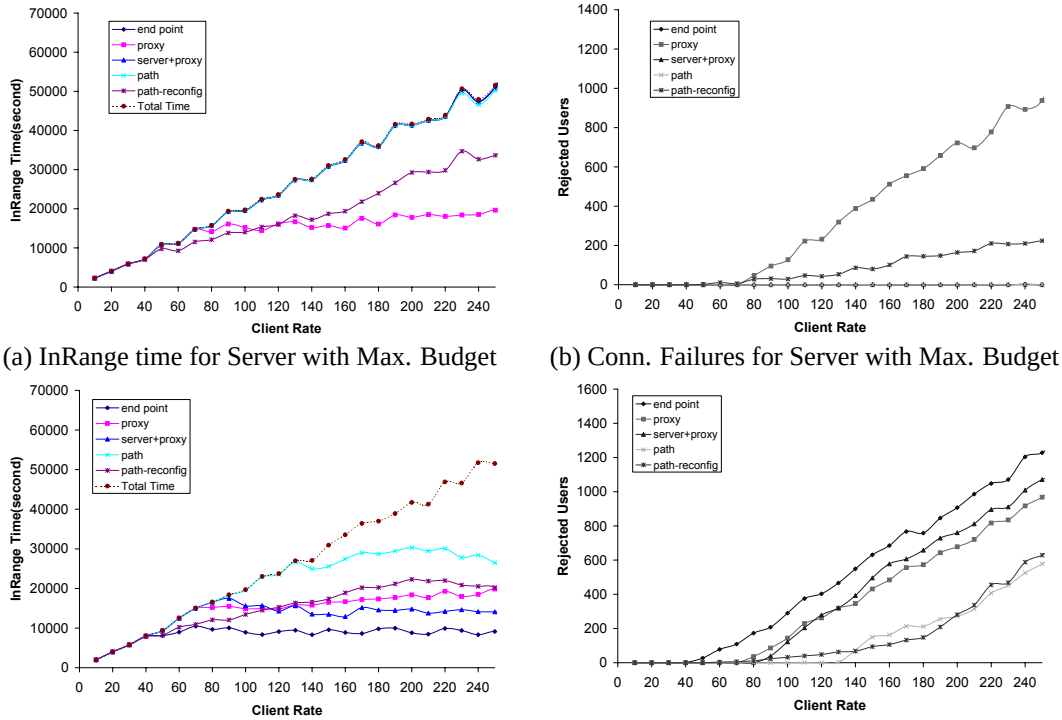
Figure 9.15: Performance of Different Client Classes under Uniform Load Distribution.

servers to benefit from spare computation resources elsewhere.

The performance of the server+proxy approach falls between the path-based approach and the end-point approach, which verifies that allowing adaptation to happen on even one more node in the middle of the communication path can improve overall performance.

Performance of Different Clients

Figures 9.15(a) and (b) show the performance (InRange time (normalized with respect to the total session time)) of the adaptation approaches from the perspective of different client classes, i.e., clients connected to the network with sufficient bandwidth versus those that use weaker connections. We can observe that while the proxy approach exhibits a more or less uniform behavior, the end-point approach demonstrates considerable preference for clients with better connectivities over others. The path approach, in addition to providing the best performance, uniformly supports different classes of clients until one runs out of computation resources beyond a certain load level. At this point, all approaches end up rejecting more clients with weak connectiv-



(a) InRange time for Server with Max. Budget (b) Conn. Failures for Server with Max. Budget

(c) InRange time for Server with Min. Budget (d) Conn. Failures for Server with Min. Budget

Figure 9.16: Performance of Different Server Classes under Uniform Load Distribution.

ity because they require more computation (image filtering and resizing operations) along the paths.

Performance of Different Server Sites

Comparing between Figures 9.16(a)–(d) allows us to draw conclusions about how the different adaptation approaches perform from the perspective of connections targeting servers with higher or lower computation budgets. The results indicate that the end-point approach shows a distinct bias, performing much better with the largest server than with the smallest one. The proxy approach performs uniformly with both servers, primarily because all computation resources are aggregated at proxy sites. The path-

based approach performs as well as the end-point approach for the largest server, and performs the best for the smallest server. This can again be explained by the flexibility brought by resource sharing and being able to use upstream nodes to do adaptation.

Another point deserving mention is the performance decrease of the server+proxy approach in Figure 9.16(c) for client arrival rates higher than 90 connections/second. This can be explained as follows: after load increases to the point where the smallest server runs out of computation resources, other server nodes continue to support filtering or resizing components because they have additional computation capacity. Since compressed connections (with components) consume less bandwidth than uncompressed ones, accepting more compressed connections for these servers can in turn decrease the number of uncompressed connections to the smallest server because the size of resource shares in the core network links shrinks as more compressed connections join in. Consequently, for the smallest server, the InRange time drops and the number of connection failures increases as load increases. Note that the path-based approach avoids this situation by exploiting resource pooling at server-side proxies.

Performance Impact of Dynamic Reconfiguration

The plots in Figure 9.14 also show that there is a considerable performance penalty incurred for disallowing reconfiguration after the path has been created. This validates the need for a *reactive* mechanism to cope with dynamic changes. In general, different types of paths may have different requirements on network resources (e.g. some of them may require more bandwidth while others may need more computation). As load changes, it is necessary to adjust allocated shares of existing paths in order to

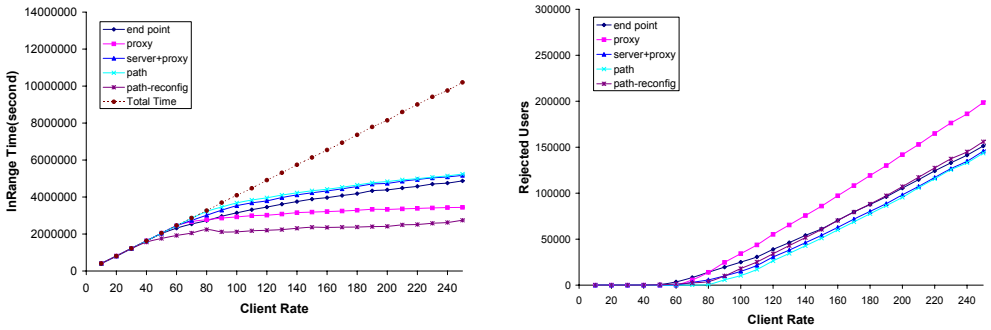
accept more connections.[7] Without reconfiguration support, adjustments for one path may end up pushing other paths out of the required range, and thereby negatively impact the overall performance.

Another detail that should be mentioned about the path-reconfig approach is the ramp-up at the end of Figure 9.14(c). This can be explained as follows: as the number of client connections increase, the number of partitions of network resources grows while decreasing the size of each partition. Eventually, it becomes difficult for clients who use weak connections to successfully connect to servers because the partition of computation resources is too small to perform the required image filtering and/or resizing operations. As a result, a large number of such connections end up getting rejected. On the other hand, connection requests from clients with higher bandwidth links continue getting accepted. Moreover, because more compressed paths are rejected, the likelihood that an uncompressed path will get pushed out of the required range decreases. This results in increased normalized InRange time for clients who use T3/T1/ADSL links.

9.5.4 Performance under Non-Uniform Load Distribution

This configuration examines how different adaptation approaches perform when connection requests from clients are directly non-uniformly towards servers. Similar to load patterns observed on the Internet, we assume a "hot-spot" model, where a small portions of servers (the hot-spots) receive most of the requests from clients. Specifically, 20% of the servers receive 80% of the total requests. We further ensure that the

[7] One can argue that using reservations may eliminate the need for dynamic adjustments, but such approaches usually have poor throughput (sustain fewer connections) as load dynamically changes

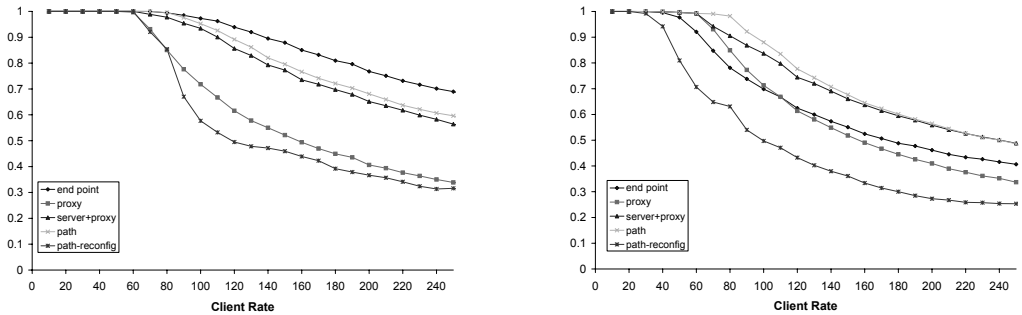


(a) Aggregate InRange Time (b) Aggregate Connection Failures

Figure 9.17: Aggregate Performance under Non-Uniform Load Distribution.

average load of large sites (i.e., sites with an OC12 link and computation budget uniformly distributed in the range [100,200)) is about 4 times the average load of small sites (i.e., sites with an OC3 link and computation budget uniformly distributed in the range of [10,100)).

Figures 9.17–9.19 show the performance achieved by the different approaches. The organization of the plots is similar to that seen earlier in the previous section. There are several observations that one can make here. First, focusing on aggregate performance, we see that the overall ranking of performance among these adaptation approaches remains the same as in the uniform distribution case. However, the total InRange time is noticeably lower than the values we saw in Section 9.5.3. This is expected because the overloaded hot-spot servers cause increased connection failures. Second, the relative performance of the path-reconfig approach is worse than seen earlier. This verifies our intuition that such an approach performs poorly when some portions of the network get overloaded; due to the absence of reconfiguration, existing paths cannot be adjusted to take advantage of surplus resources in network regions that are lightly loaded.



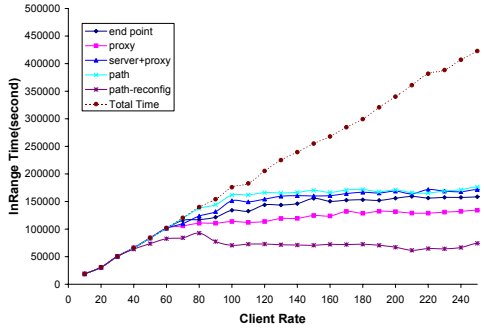
(a) Normalized InRange Time for T3/T1/ADSL Clients

(b) Normalized InRange Time for Dialup/Wireless Clients

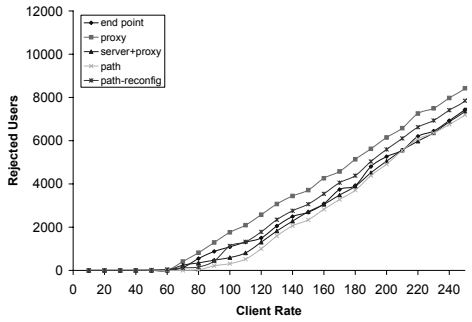
Figure 9.18: Performance of Different Client Classes under Non-Uniform Load Distribution.

Looking at performance seen by clients with different connectivity options, the overall trends mirror those seen for uniform traffic. Figure 9.18(a) is a little different from the corresponding plot in Figure 9.15(a) in that the end-point approach gets the highest normalized in-range time for clients using T3/T1/ADSL connections. This value comes at the cost of more connection failures for clients with weak connections (recall that 75% of all clients use dialup/wireless connections). The aggregated InRange time of the path-based approach is still the best among the five approaches.

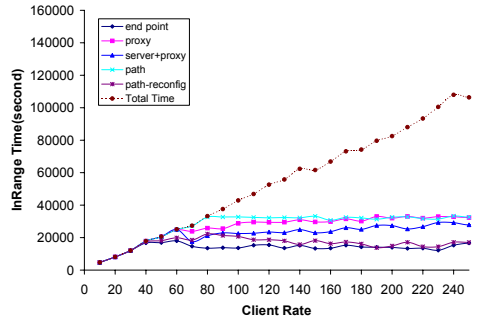
Looking at the performance from the perspective of servers with the maximum and minimum computation budgets (Figure 9.19, it can be observed that the path-based approach outperforms all other approaches. This again verifies the benefit of resource sharing in the network: overloaded sites can always take advantage of spare computation resources elsewhere. This is true even for sites that have a large amount of computation resources, because there will be a load level that causes these sites to become overloaded. The end-point approach performs poorly on sites with smaller computation budgets. The proxy approach exhibits the same behavior, independent of computation budget, as in the uniform distribution case. However, as before, the



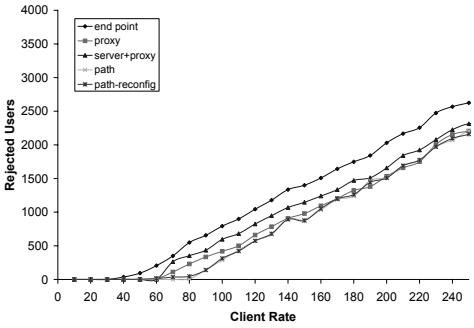
(a) InRange time for Server with Max. Budget



(b) Conn. Failures for Server with Max. Budget



(c) InRange time for Server with Min. Budget

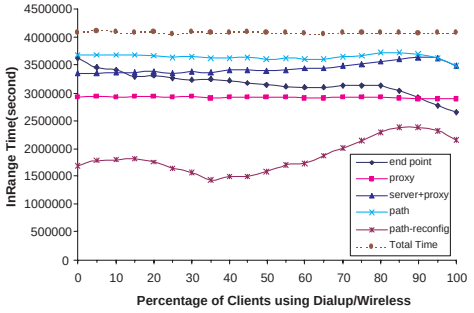


(d) Conn. Failures for Server with Min. Budget

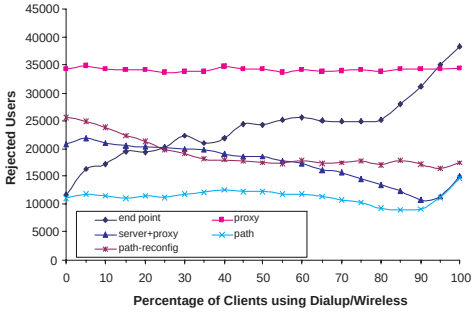
Figure 9.19: Performance of Different Server Classes under Non-Uniform Load Distribution.

problem of bandwidth waste results in the network core becoming an early bottleneck as load increases.

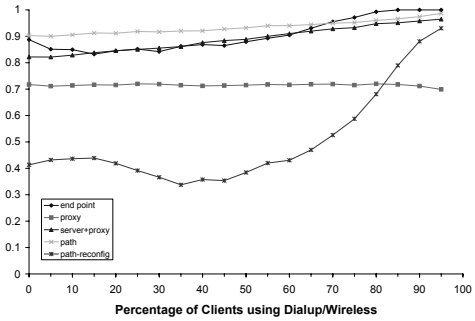
A more interesting point with this set of results is that they used the same resource distribution between server and ISP nodes as in Section 9.5.3, namely one that *assumes a uniform load distribution*. This is important because load distributions at run-time are likely to be different from what is considered when deciding about how to provision resources in the network. Our results show that the path-based approach still performs very well even with an inaccurate knowledge of load distribution. This robustness mainly comes from the shared resource pools across the whole network



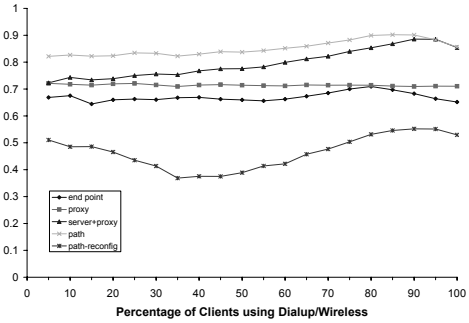
(a) Aggregate InRange Time



(b) Aggregate Connection Failures



(c) Normalized InRange Time for T3/T1/ADSL Clients



(d) Normalized InRange Time for Dialup/Wireless Clients

Figure 9.20: Performance under Different Client Connectivity Profiles.

that act like “buffers”, absorbing most negative impact because of the unexpected load.

9.5.5 Performance under Different Client Connectivity Profiles

In this configuration, we examine how the different adaptation approaches perform when different fractions of clients use different connectivity options. The simulations run with the same settings as in Section 9.5.4 with only two differences: the client arrival rate was fixed at 100 users per second, and we varied the percentage of clients that use weak connections (dialup or wireless) from 0 to 100 percent (the ratio between numbers of clients that use dialup and wireless connections was maintained at

1:2).

Figure 9.20(a)–(d) shows the performance results. One can observe that among the four approaches with reconfiguration support, the end-point approach is the only one that exhibits decreasing performance as more clients use weak connections while the other three approaches achieve relatively stable performance across different configurations. Because the end-point approach does not support resource sharing, smaller sites or overloaded sites end up rejecting many connection requests once they run out of computation resources.

It can also be seen that the path-reconfig approach performs better when the client connectivity profile is more uniform. This can be explained as follows: as more paths exhibit similar behavior (i.e., have similar resource requirements), there is lower likelihood that an existing path will get pushed out of its required performance range by the arrival of a new connection. Stated differently, the more heterogeneous the environment, the larger the need for dynamic reconfiguration.

Some clarification is needed for the increasing InRange time achieved in Figure 9.20(a) by the server+proxy approach as more clients use weak connections. While this may appear counter-intuitive, the following explains this behavior. Consider what happens when clients use connections that have sufficient bandwidth. As load increases, initially modest compression (filtering/resizing) will be introduced into the paths and executed on the server sites. As the number of connections further increases, the size of partitions on the server sites will eventually become too small to do the required compression. Consequently, after this point, the network core starts become a bottleneck and once it does, new connections end up getting rejected. Note however that when this happens, the proxy sites close to clients remain underutilized

because they are ineffective for reducing bandwidth requirements in the network core.

On the other hand, the situation is different when most of the clients are using weak connections. Due to the limited bandwidth of weak connections, strong compression will be required at the server sites from the beginning. The strong compression results in considerable saving in bandwidth in the network core. Therefore, as load increases, some of the new connections can take advantage of the saved bandwidth in the network core and do compression at the client side proxy sites. As a result, the utilization of the proxy sites is high and more connections are accepted.

The above behavior also provides further evidence for the benefits by using additional nodes in the data path to perform adaptation operations.

9.5.6 Summary of Simulation Results

The main results from our study are summarized below:

1. Support for dynamic reconfiguration is important for the performance of both individual paths and the whole network.
2. The end-point approach usually works well with server sites that have a large amount of computation resources and for clients that connect to the network with relatively high bandwidth links. However, servers that have limited computation capacity or clients that use weak connections may suffer from poor performance using such an approach.
3. The proxy approach usually does not exhibit bias towards different types of servers or clients. The shared resource pool at proxy sites can bring better performance for small server sites or clients that have weak connectivity. However,

constraining the adaptation to only occur before the last hop can cause considerable resource wastage in the network, in turn leading to early saturation as load increases.

4. The path-based approach has all the benefits of both end-point and proxy approaches. Adaptation can be conducted on upstream nodes without being limited to the node before the last hop. More importantly, the approach sets up shared resource pools across the whole network, providing the most flexibility for overloaded servers to benefit from spare computation resources elsewhere. In summary, with effective resource management strategies, this approach provides the best and the most robust performance under different network configurations.

9.6 Summary

In this chapter, we have presented an extensive evaluation of our framework, under different network configurations and using different applications. We carried out our experiments by running typical applications on top of the CANS infrastructure, and simulating our schemes for large-scale networks. The experimental results validate our approaches, verifying that:

- Network awareness in data communication can be provided to regular applications by injecting application specific functionality into the network and letting the underlying infrastructure control such paths.
- Network-aware communication paths created with our automatic path creation strategies provide applications with considerable performance advantages.

- Fine tuned, desirable, and continuous adaptation behaviors can be constructed using our strategies for path creation and reconfiguration.
- The run-time overheads of the CANS infrastructure are negligible, and reconfiguration cost is small for most applications, and can be further substantially reduced by our local mechanisms.
- Compared with adaptation using end-point or proxy-based approaches, CANS-like path-based approach provides the best and the most robust performance for different servers/clients under most network conditions.

Chapter 10

Summary and Future Work

In this chapter, we summarize the work presented in this dissertation, discuss future work, and our perspective on the longer term implications of this work.

10.1 Summary

We observe that network awareness in data communication is important for accessing services across the Internet. In addition to transmitting data like a conventional communication path, a network-aware communication path is capable of automatically and continually adapting to different underlying network conditions according to application requirements. This dissertation proposes a general framework for providing various applications with network-aware communication paths.

To continually match application requirements with underlying network conditions, application specific functionality, organized in the form of components with a well-defined interface, is dynamically injected into the communication path; the

underlying infrastructure is used for creating and controlling such communication paths and managing resources across the network. The former (application specific functionality) allows applications to customize its data communication requirements, while the latter (underlying infrastructure) provides common support for realizing network awareness. Combining these two together, regular (legacy) applications can easily be augmented with adaptation capabilities, requiring only high-level input from applications.

Compared with end-point or proxy-based approaches, network awareness in our framework is realized throughout all (possible) network resources along communication paths.

To build a path-based infrastructure to support network-aware communication paths, several challenging problems need to be addressed before this vision could become reality. These problems were the focus of this dissertation. In particular, we have presented solutions for the following previously *open* questions: 1) how to model and organize application specific functionality so that adaptation operations can be separated from other parts of the application, and control logic of communication paths can be extracted and built into the underlying infrastructure? 2) How to automatically construct the *best* path based on application requirements and network conditions? 3) How to efficiently modify such paths when network conditions change? 4) How to efficiently manage network resources across the network? Solutions for these problems are indispensable for any practical deployment of a path-based infrastructure.

Below, we briefly review our key schemes.

Dynamic Composition and Type-Based Modeling In our framework, application specific functionality is organized in the form of components with a well-defined interface. To separate path creation/adaptation logic from the application itself, our framework constructs communication paths by dynamically composing different components together. Dynamic composition is supported by the component interface and a high-level type specification of component behaviors. The use of the component paradigm make our framework highly extensible: the functionality contained in the infrastructure grows as new components being added in; besides, application development is completely independent from component authoring, so is the latter from that of other components. More importantly, such a composition view lays a solid foundation for extracting common logic for creating and controlling network-aware paths for inclusion at the infrastructure level.

Automatic Path Creation Strategies In our framework, network-aware communication paths are created using *automatic path creation strategies*. Our strategies can produce communication paths with optimized performance in accordance with application requirements and underlying network conditions, requiring only high-level information from applications. Such automatically generated paths, in addition to providing applications with considerable performance benefits (i.e. throughput, latency, or data quality etc.), can also address other requirements such as required data format (when different from that of the data source), security guarantees etc., which are effected by the characteristics of the network resources along the path.

Our path creation strategies are very flexible in that they can be used with applications with very different performance requirements, i.e., some may need to max-

imize/minimize some performance metrics while others may demand the guarantee of some performance metric being in a specific range with other performance metrics optimized. The calculation of such network-aware communication paths does not require a centralized entity or global knowledge (except for commonly used types) across different network domains, and can be built incrementally in a distributed fashion. Furthermore, in addition to building a whole new path, our path creation strategies can also be used to replace small portions of an existing path while maintaining some overall performance guarantees. This is very important for our vision of network awareness: every segment of a communication path can continually adapt to changes in the network, independently and concurrently.

Dynamic Path Reconfiguration To provide applications with dynamic adaptation, our framework includes system support for *low-overhead dynamic path reconfiguration*. Path reconfiguration is controlled completely by the underlying infrastructure so that the application can concentrate on its own "business" logic.

The reconfiguration process is quite flexible in that different applications are allowed to customize different levels of semantic continuity guarantees for data transmissions when reconfiguration occurs. In addition to modifying the whole communication path (which is called global reconfiguration), our reconfiguration strategies also support independent and concurrent modification of small portions of the path (called local reconfiguration). When network conditions change, local reconfigurations will be tried first, with global reconfiguration being triggered only if local reconfiguration cannot effectively cope with the change. Such a multiple-level reconfiguration is not only important for adaptation agility but also for use of such an infrastructure

with long communication paths, which usually span multiple network administrative domains.

Combining the support for path creation and reconfiguration together, our framework provides various applications with fine-tuned, desirable adaptation behaviors for dynamic changes in the network. Since when and how to adapt is completely controlled by the underlying infrastructure, network-oblivious applications can be augmented with adaptation capabilities.

Distributed Resource Management Deploying such a path-based infrastructure requires a large network of infrastructure-enabled nodes that overlay on existing Internet infrastructure to run computation required by these augmented paths. Our framework includes distributed strategies for managing network resources among different communication paths and different network regions. By efficiently allocating and adjusting resource shares of multiple communication paths, our strategies provide individual paths with good performance and improve the throughput of the whole network, i.e., increase the number of connections can be sustained. Moreover, our framework contains a hierarchical model and a corresponding algorithm to set up shared resource pools across the network. By distributing computation resources across different network domains, the overall performance of the whole network gets improved because overloaded network regions can make use of shared resources from others.

The CANS infrastructure Our framework is realized as a Java-based programmable network infrastructure called Composable Adaptive Network Services (CANS). CANS is built from the ground up to provide applications with network-aware communica-

tion paths. Extensive experiments have been carried out with different applications running on top of CANS. The results validate the effectiveness of our schemes.

10.2 Conclusion

Our framework provides a complete set of solutions for building a path-based infrastructure that provides applications adaptation capability to changes in the network using network-aware communication paths. By building these solutions into a programmable network infrastructure (CANS) and with extensive experiments using typical Internet applications, we have verified that

- Automatic path creation and reconfiguration are achievable and do in fact yield substantial performance benefits.
- Our approach is effective at providing applications that have different performance preferences with fine tuned, desirable adaptation behaviors.
- Despite the flexibility, the overhead incurred by the CANS infrastructure is negligible, and the cost to reconfigure data paths is acceptable for most applications. Additionally, these costs can be further reduced substantially using local planning and reconfiguration mechanisms.
- The resource management strategies are effective in improving both individual path performance and resource utilization of the whole network.
- Compared with alternative end-point and proxy-based approaches, using a CANS-like path-based approach to realize network awareness throughout the entire communication path not only results in better responsiveness to changes in the

network, but more importantly provides the flexibility of adaptation anywhere and resource sharing across the network, allowing overloaded network regions to take advantage of spare resources from other parts of the network. This makes such a path-based approach the best and the most robust way for delivering satisfactory performance across the network.

In conclusion, this dissertation has presented a general framework that provides applications with network-aware communication paths. These paths, which are automatically created by the underlying infrastructure, can further continually adapt to dynamic changes in the network. To the best of our knowledge, our work is among the first providing such network awareness in the context of a general framework.

10.3 Future Work

There are two issues that have not been completely integrated into our framework: security concerns and resource monitoring across the network. These two problems are relatively independent of the focus (on network awareness) of this dissertation in that the support for distributed authentication, secure execution of mobile code, and information about resource availability required by our framework are likely to be important features of other distributed systems as well. There is already a large body of literature on these topics, and several proposals. In the near future, the work described in this dissertation can be extended as follows.

10.3.1 Security Concerns

Security concerns are raised by the need for deploying and executing mobile code along communication paths that span multiple administrative network domains. This requires distributed authentication mechanisms as well as a secure execution environment for mobile code. Distributed authentication (and trust management in general) frameworks (such as PolicyMaker [8], KeyNote [7], Taos [56], and dRBAC [20] etc.) allow users to express distributed trust relationships, such infrastructures can be integrated to control code downloading in our framework.

The current implementation of the CANS execution environment provides a secure environment for running downloaded code by leveraging the the features of Java programming language and existing security features built into JVMs [46]. By using a custom class loader with an environment object (via which drivers can only access the functionality of the CANS EE), execution of components from different locations can be effectively isolated from each other. The passive interface of the CANS Driver further makes it relatively simple to constrain how many resources can be consumed by a particular path.

For components embedded with native code (e.g. via the JNI interface), the situation is more complicated. One way to control native code is by intercepting the JNI interface and employing a similar controlling strategy as the commonly used sandboxing technique.

10.3.2 Resource Monitoring Utility

Our framework also needs a resource monitoring utility that provides information about dynamic resource availability. While our distributed/local versions of the planning and reconfiguration strategies greatly reduce the requirements for global information of resource availability information, we still need efficient mechanisms for monitoring network resources in a wide area network. Furthermore, effective filtering mechanisms are also needed to reduce unnecessary path reconfiguration. The approaches being evaluated in existing and proposed frameworks such as Remos [16], Network Weather Service [57], Grid Monitoring Services [61], and [37] may be used with our framework.

10.4 Perspective

Our approach of building network-aware communication paths by dynamically and automatically composing and managing components reveals the feasibility of integrating and orchestrating between diverse functionality across the network. By composing functionality from different sites to address user's high level requirements, individual services are no longer isolated islands that implement simple functions in a monolithic way. Instead, the world of services is interconnected and new services can be constructed on the fly as appropriate for the needs of different users. This vision presents users with a network characterized by truely integrated functionality. The approach advocated in this dissertation takes this view and provides solutions for how to model, compose, and control various functionality to meet user requirements on data communication, providing automatic adaptation to regular applications in dynamic

environments.

Our approach is based on a data flow view that models functionality as a mapping between input data and output data type while maintaining the same underlying semantics for the processed data. Such a view is quite effective for various transformation services. However, to support compositions among arbitrary services, this view may need to be extended for modeling behaviors of more complicated components or services that can change the semantics of the processed data. In turn, it is likely that the semantic description of such services may need to be extended to enable composition. We look at this as a higher level problem, which can be built on top of our framework with new models to describe and deduce semantic information, possibly leveraging standard ontologies such as being developed by the Semantic Web [4] and IEEE's Standard Upper Ontology [27] efforts.

The emergence of industry standards (WSDL [11] UDDI [13]) for describing and searching components across the Internet reflects the increasing need for interoperability across the Internet. We believe that the prospect of intelligence in the network will eventually become reality, allowing seamless integration of functionality in the Internet to meet various user needs. As such needs grow, underlying infrastructure will be required to provide network-aware communication, and efficient and seamless composition of such functionality. We view the work presented in this dissertation as a step toward this direction and look at the infrastructure that supports construction of intelligent applications by composing functionality across the Internet as the longer-term outcome of this work.

Appendix A

Component Profile Information

Table A.1 lists some of the profiled values of *computation load factor* (`load`, i.e., number of operations for each byte of incoming data) and *bandwidth impact load factor* (`bwf`) of the components used in this dissertation. Table A.1(a) lists these values of drivers that have only a single configuration: `zip`, `unzip`, `Encryption` and `Decryption`. The data used in the profiling is a typical HTML page with the size of 14 K bytes. Table A.1(b), (c) show the values of `ImageFilter` and `ImageResizer` respectively, both of these drivers support multiple configurations. The profiling used a typical high-quality JPEG image with the size of 25K bytes (profiling results using images of medium and poor qualities are omitted for brevity).

A.1 Profiling with different data sizes

We model the performance characteristics of a driver c using its *computation load factor* ($\mathrm{load}(c)$), the average per-input byte cost of executing the component, and the

Driver	`load` (op/byte)	`bwf`
`Zip`	0.133	0.3175
`Unzip`	0.118	3.15
`Encryption`	0.435	1
`Decryption`	0.435	1

(a) Parameters of Single Configuration Drivers

Configuration	1	2	3	4	5	6	7	8
Resizing	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
`load` (op/byte)	2.786	3.070	3.806	4.316	4.674	5.301	6.295	6.939
`bwf`	0.121	0.195	0.287	0.389	0.488	0.606	0.710	0.847

(b) Parameters of ImageResizer

Configuration	1	2	3	4	5	6	7	8	9
Image Quality	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
`load` (op/byte)	2.552	2.590	2.602	2.642	2.652	2.733	2.746	2.707	2.771
`bwf`	0.271	0.402	0.531	0.574	0.751	0.87	0.961	1.055	1.411

(c) Parameters of ImageFilter

Table A.1: Profiled Parameter of Components

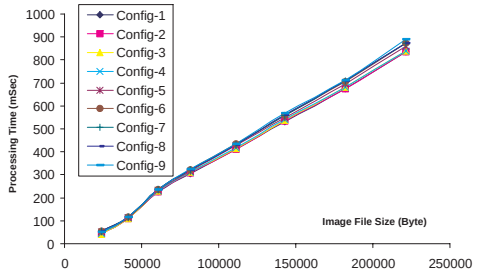
bandwidth impact factor ($\mathrm{bwf}(c)$), the average ratio between input and output data volume. This linear model is based on profiling we have conducted with various components. Here, we present a subset of these profiling results using the `ImageFilter`

and the `ImageResizer` components. For this experiment, we profiled the performance of `ImageFilter` and the `ImageResizer` with a set of image files of sizes ranging from 21K bytes to 221K bytes.

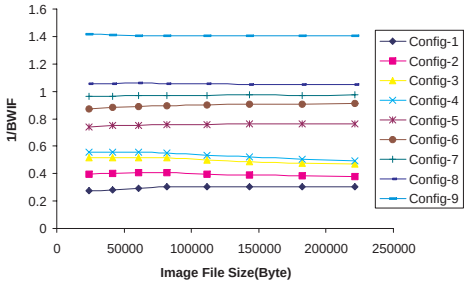
Figure A.1 shows the computation time of these two components for processing these images. It shows that the computation time is basically linear with the size of input data, and the values of `bwf` remain almost constant with only small variations. Though there could be some unusual components that exhibit different performance characteristics, whose computation load is not linear with the size of incoming data, the result validates that (load(c) and bwf(c) are reasonable approximations for modeling performance characteristics of regular components.

A.2 Profiling Component Composition

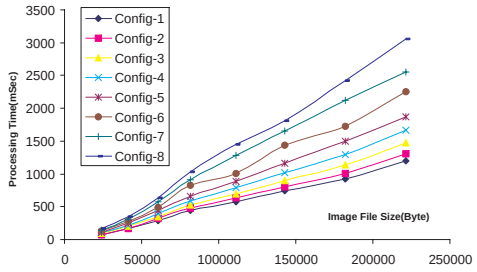
We also need to verify that the linear approximation remains valid when multiple components are execution together in the same execution environment. To observe the performance under component compositions, we profiled the computation time of the composition of `ImageResizer` and `ImageFilter`, both of which are configured to use the 5th configuration. The profiling used the same set of image files as the previous experiment. Figure A.2 shows the profiled result, and compares it with the expected values calculated using our approach: The expected value is calculated using $s \cdot (\mathrm{load}(\mathrm{Resizer}_5) + \mathrm{bwf}(\mathrm{Resizer}_5) \cdot \mathrm{load}(\mathrm{Filter}_5)$, where s is the input image size. This figure verifies that the calculated values using our approach are very close to the actual values we measured, thus verifying the linear model works for composition.



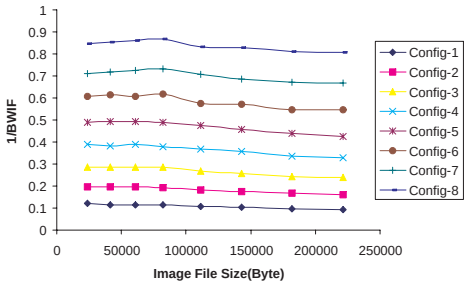
(a) `Processing Time` of `ImageFilter`



(b) `bwf` of `ImageFilter`



(a) `Processing Time` of `ImageResizer`



(b) `bwf` of `ImageResizer`

Figure A.1: Profiles with different data sizes.

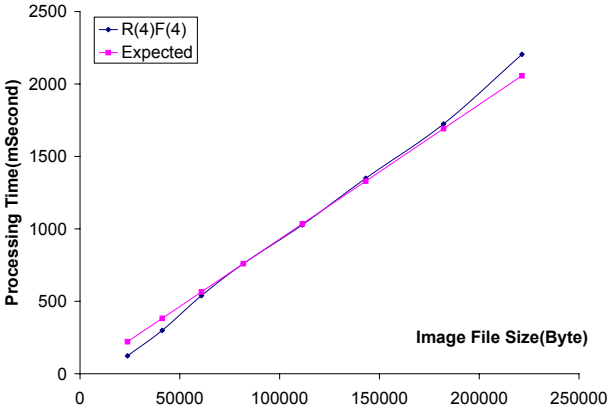


Figure A.2: Computation time of a component Composition (`ImageResizer`(5)–`ImageFilter`(5)).

Appendix B

Emulating Real Network Behaviors Using Sandboxing

To investigate the performance of our framework in a wide range of network configurations, in our experiments described in Chapter 9, we extensively used a sandboxing toolkit to emulate different network conditions. The sandboxing toolkit [10] (designed by Chang et al.) can be used to control resource consumption of applications by intercepting system calls between the applications and the underlying operating system. Taking the example of network resources, the sandboxing toolkit emulates links with different bandwidth properties by constraining the rate at which applications are allowed to send/receive data. Experimental results presented in [10] verify that this toolkit can effectively control application usages of CPU, memory, and network bandwidth.

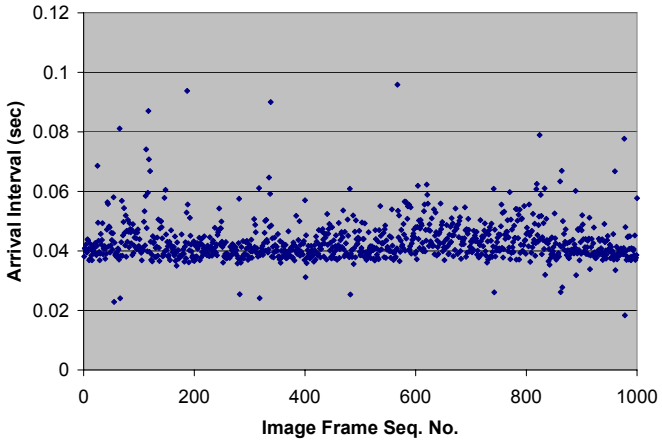
However, because the purpose of our experiments is to study adaptation behaviors of applications, the effectiveness of the emulation with the sandboxing toolkit needs

to be studied for validating our experimental results. This is especially the case for some of our experiments, where we used the sandboxing toolkit on a local network to emulate some "logical" network links that may involve multiple hops (e.g. the connection between the edge server and the proxy server in Section 9.1).

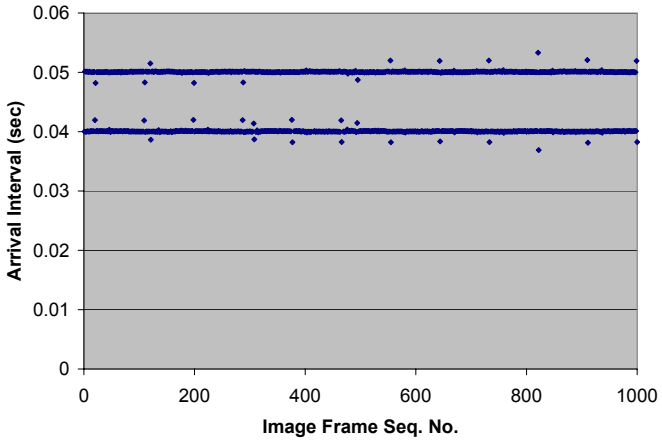
To examine the differences between the behaviors of a real network configuration and the behaviors emulated using the sandboxing toolkit, we conducted an experiment, using the image streaming application described in Chapter 9. In this experiment, we compared the behavior of the application when running in a real network configuration with that on the emulated one (i.e. in a LAN with the sandboxing toolkit). For the real network configuration, we run the image client on a laptop using a wireless network (IEEE802.11b) in our lab, downloading images from an image server running on elsewhere on campus. Including the last wireless connection, there were a total of 6 hops between these two machines [1]. For the emulation, we run the application in a 100Mbps Ethernet LAN connected with a Ethernet switch, and emulated the behaviors of the real network configuration by using the sandboxing toolkit to constrain data transmission between the server and the client applications to correspond to the measured bandwidth value in the real configuration.

The bandwidth measured between these two nodes in the real network configuration was 542KBps. The average size of the image files used in this experiment was 24K bytes. The measured throughout in the real network configuration was 21.87 frames per second. The emulation provided a throughput of 22.30 frames per second, validating that the sandboxing toolkit is effective in emulating the overall performance characteristics of a real network configuration.

[1] Determined by using the *traceroute* utility



(a) Real Network Configuration



(b) Emulation with the sandboxing toolkit

Figure B.1: Arrival Interval of Individual Image Frames

To further examine the microscopic behaviors of these two cases, we recorded the arrival interval of individual image frames. The result is shown in Figure B.1. In this Figure, the x-axis denotes the image frame sequence numbers, and the y-axis denotes the interval between the arrival times of two consecutive frames. Figure B.1(a) shows that the interval values in the real network configuration are clustered around 0.04 to 0.05 seconds. Figure B.1(b) shows that in the emulation, the values of arrival

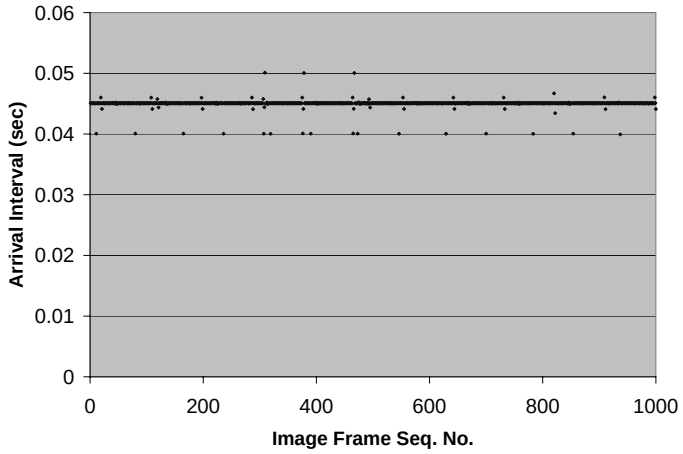


Figure B.2: Averaged Arrival Interval Time for Every Two Adjacent Image Frames

intervals are clustered in two groups: about half of the frames have a value of 0.04 seconds, and the other half a value of 0.05 seconds. This behavior is mainly due to the accuracy limitation of the fine-grained timer used by the sandbox implementation, which can only support accuracy at a granularity of 10 milliseconds. To account for this implementation artifact, we examined the average of the interval values for every 2 adjacent frames ($t_i = (t_i + t_{i+1})/2$). The result is shown in Figure B.2, which shows the averaged value of arrival interval is 0.045 seconds, a close match to that seen on the real network.

Our results show that despite the behaviors emulated by the sandboxing toolkit not being exactly the same as that on a real network configuration, the former provides a very close approximation. Therefore, we conclude that for the specific experiments undertaken in this dissertation, the use of the sandboxing toolkit should not affect the conclusions drawn from our experimental results.

Bibliography

[1] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The switchware active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29 – 36, 1998.

[2] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *Proc. of the SIGCOMM'98*, August 1998.

[3] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles(SOSP)*, October 2001.

[4] Tim Berners-Lee. Services and semantics: Web architecture. In *http://www.w3.org/2001/04/30-tbl.html*, 2001.

[5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating*

Systems Principles (SOSP '95), Copper Mountain Resort, Colorado, December 1995.

[6] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. v. Renesse, and W. Vogels. The Horus and Ensemble Projects: Accomplishments and Limitations. In *the DARPA Information Survivability Conference and Exposition (DISCEX '00)*, January 2000.

[7] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[8] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of IEEE Conf. on Privacy and Security*, 1996.

[9] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, October 2002.

[10] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-Constrained Sandboxing. In *Proc. of the 4th USENIX Windows Systems Symposium*, August 2000.

[11] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. Technical Report W3C Note 15 March 2001, W3c, march 2001.

[12] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM Sigmetrics*, pages 1–12, Santa Clara, CA, June 2000.

[13] UDDI Spec Technical Committee. UDDI version 3.0. Technical report, www.uddi.org, Jul 2002.

[14] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of ACM SIGCOMM'98*, 1998.

[15] A. DeSoto. Using the beans development kit 1.0, a tutorial. Technical report, Sun Microsystems, Sep 1997.

[16] A. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. Remos: A resource monitoring system for network-aware applications. Technical Report CMU-CS-97-194, Carnegie Mellon School of Computer Science, 1997.

[17] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[18] A. Fox, S. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Prespectives. *IEEE Personal Communication*, August 1998.

[19] A. Fox, S. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, October 1997.

[20] E. Freudenthal, T. Pesin, E. Keenan, L. Port, and V. Karamcheti. dRBAC: Distributed Role-Based Access Control for Dynamic Coalition Environments. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[21] S. D. Gribble and et al. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of IEEE Computer Networks on Pervasive Computing*, 2000.

[22] S. D. Gribble, M. Welsh, E.A.Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructual Services. In *Proc. of the 1999 Usenix Annual Technical Conf.*, June 1999.

[23] AN NodeOS Working Group. NodeOS Interface Specification. Technical report, January 2001.

[24] Network Working Group. Next steps for the IP QoS architecture. In *RFC: 2990*, 2000.

[25] Network Working Group. Multiprotocol label switching (mpls) traffic engineering management information base. In *Internet Draft*, November 2002.

[26] Network Working Group. Resource ReSerVation Protocol (RSVP). In *RFC: 2990*, 2205.

[27] Standard Upper Ontology (SUO) Working Group. IEEE Standard Upper Ontology Scope and Purpose. In *http://suo.ieee.org/scopeAndPurpose.html*, 2001.

[28] J. Haartsen. BLUETOOTH– The universal radio interface for ad hoc, wireless connectivitity. *Ericsson Review*, 1998.

[29] M. Henning and S. Vinoski. *Advanced CORBA(R) Programming with C++*. Addison-Wesley Pub Co, Feb 1999.

[30] M. A. Hiltunen and R. D. Schlichting. The Cactus Approach to Building Configurable Middleware Services. In *the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, October 2000.

[31] G. Hunt. Detours: Binary interception of win32 functions. In *Proc. of the 3rd USENIX Windows NT Symp.*, Settle, WA, July 1999.

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

[33] University of Southern California Information Sciences Institute. Internet protocol. In *RFC: 791*, 1981.

[34] University of Southern California Information Sciences Institute. Transmission control protocol. In *RFC: 793*, 1981.

[35] A. D. Joseph, J. A. Tauber, and M. F. Kasshoek. Mobile Computing with the Rover Toolkit. *IEEE Transaction on Computers:Special Issue on Mobile Computing*, 46(3), March 1997.

[36] E. Kiciman and A. Fox. Using Dynamic Mediation to Intergrate COTS Entities

in a Ubiquitous Computing Environment. In *Proc. of the 2nd Handheld and Ubiquitous Computing Conference (HUC'00)*, March 2000.

[37] M. Kim and B. Noble. Mobile network estimation. In *Proceedings of the Seventh ACM Conference on Mobile Computing and Networking*, July 2001.

[38] A. Mallet, J. Chung, and J. Smith. Operating System Support for Protocol Boosters. In *Proc. of HIPPARCH Workshop*, June 1997.

[39] Sun Microsystems. Enterprise javabeans(tm) specification 2.1 proposed final draft 2. Technical report, Jun 2003.

[40] R. Mohan, J. R. Simth, and C.S. Li. Adapting Multimedia Internet Content for Universal Access. *IEEE Transactions on Multimedia*, 1(1):104–114, March 1999.

[41] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, 1999.

[42] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996.

[43] A. Nakao, L. Peterson, and A. Bavier. Constructing End-to-End Paths for Playing Media Objects. In *Proc. of the OpenArch'2001*, March 2001.

[44] B. Noble. System Support for Mobile, Adaptive Applications. *IEEE Personal Communications*, pages 44–49, February 2000.

[45] Brian D. Noble. *Mobile Data Access*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[46] Scott Oaks. *Java Security (2nd Edition)*. O'Reilly Associates, may 2001.

[47] B. Raman, R.H. Katz, and A. D. Joseph. Universal Inbox: Providing Extensible Personal Mobility and Service Mobility in an Integrated Communication Network. In *Proc. of the Workshop on Mobile Computing Systems and Applications (WMSCA'00)*, December 2000.

[48] P. Reiher, R. Guy, M. Yavis, and A. Rudenko. Automated Planning for Open Architectures. In *Proc. of OpenArch'2000*, March 2000.

[49] L. Subramanian, I. Stoica, H. Balakrishnan, and Randy Katz. OverQoS: Offering Internet QoS Using Overlays. In *Proceedings of 1st HotNets Workshop*, 2002 October.

[50] P. Sudame and B. Badrinath. Transformer Tunnels: A Framework for Providing Route-Specific Adaptations. In *Proc. of the USENIX Technical Conf.*, June 1998.

[51] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, pages pp80–86, January 1997.

[52] A. Troelsen. *Developer's Workshop to COM and ATL 3.0*. Wordware Publishing, Apr 2000.

[53] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible

Location and Transport of Wide-Area Resources. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, October 1999.

[54] U. Varshney and R. Vetter. Emerging Mobile and Wireless Networks. *Communications of the ACM*, pages 73–81, June 2000.

[55] D. J. Wethrall, J. V. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proc. of 2nd IEEE OPENARCH*, 1998.

[56] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the taos operating system. *ACM Trans. on Computer Systems*, pages 3–32, 1994.

[57] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[58] The IPSEC working group. IP security protocol (ipsec). In *Internet Draft*, April 2003.

[59] M. Yarvis. *Conductor: Distributed Adaptation for Heterogeneous Networks*. PhD thesis, UCLA, Department of Computer Science, Nov 2001.

[60] Y. Yemini and S. daSilva. Towards programmable networks. In *FIP/IEEE International Workshop on Distributed Systems*, oct 1996.

[61] X. Zhang, J. Freschl, and J. Schopf. A Performance Study of Monitoring and Information Services for Distributed Systems. In *Proceedings of Interna-*

tional Symposium on High Performance Distributed Computing (HPDC), August 2003.