

LIBRARY
Michigan State
University

This is to certify that the
dissertation entitled

SERVICE CLOUDS: OVERLAY-BASED INFRASTRUCTURE
FOR AUTONOMIC COMMUNICATION SERVICES

presented by

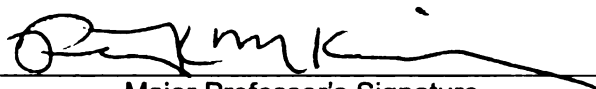
FARSHAD ALAM-SAMIMI

has been accepted towards fulfillment
of the requirements for the

Ph.D.

degree in

Department of Computer
Science and Engineering



Major Professor's Signature

11/20/07

Date

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE

**SERVICE CLOUDS:
OVERLAY-BASED INFRASTRUCTURE FOR AUTONOMIC
COMMUNICATION SERVICES**

By

Farshad Alam-Samimi

A DISSERTATION

**Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of**

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2007

ABSTRACT

SERVICE CLOUDS: OVERLAY-BASED INFRASTRUCTURE FOR AUTONOMIC COMMUNICATION SERVICES

By

Farshad Alam-Samimi

In autonomic computing, systems manage their own operation and adapt to changes in the environment, with only limited human guidance. Further, autonomic computing supports pervasive computing, which removes the traditional boundaries for how, when, and where humans and computers interact. One area in which autonomic behavior can be particularly helpful is communication-related software; autonomic computing can be used to enhance quality of service in the presence of dynamic or adverse conditions. However, coordinating the actions of software components that are distributed across multiple platforms is a complex task. Developers of autonomic communication services can benefit from a software infrastructure to support coordinated actions among system layers, and across system platforms.

In this dissertation, we introduce *Service Clouds*, which represents a new approach to supporting the development of autonomic communication services. In this approach, collaborating overlay nodes use dynamic software composition to realize autonomic distributed services. The key insight is that overlay networks provide a “*blank computational canvas*” on which services can be instantiated and configured as needed, and later reconfigured in response to changing conditions. We have implemented a prototype of Service

Clouds and evaluated it both for high-performance communication services atop the Internet, and for mobile computing services. The Service Clouds architecture and prototype enable developers to rapidly design and deploy autonomic communication services. This dissertation addresses four aspects of autonomic communication services.

First, we explore requirements for autonomic communication services, identify *constituent components* for an infrastructure. We design the Service Clouds overlay-based model and architecture and describe how to organize components, specify interactions among building blocks, and provide basic algorithms and protocols to compose services. We conclude this part with a description of the prototype.

Second, we evaluate the Service Clouds model on the Internet. We use the prototype to construct and deploy two services: *TCP-Relay*, in which a node is selected and configured dynamically to serve as a relay to expedite data transfer; and *MCON*, a multipath connection algorithm that exploits network topology to dynamically establish a high-quality secondary path, as a shadow connection to the primary path to support robust streaming.

Third, we investigate extension of Service Clouds to the wireless edge of the Internet to support pervasive mobile computing. Specifically, we address multimedia streaming and demonstrate how dynamic creation and configuration of proxy services support QoS streaming in highly dynamic conditions and frequent relocation of users.

Fourth, we address composition of distributed services atop an overlay network. We introduce and evaluate *Dynamis*, a generic algorithmic approach to locating suitable nodes on which to map service entities with minimal probing overhead.

© Copyright by
FARSHAD ALAM-SAMIMI
2007

To my loving parents.

Thank you for encouragement, support, and love!

ACKNOWLEDGMENTS¹

My advisor and guidance committee chairperson, Dr. Philip K. McKinley, supervised this work and guided me. Special thanks to him for his invaluable advice and unlimited time spent to correct my mistakes. I would like to thank other members of my guidance committee: Dr. Betty H.C. Cheng, Dr. Hayder Radha, and Dr. Sandeep Kulkarni, for their help and advice. I am grateful to researchers in the Software Engineering and Network Systems Laboratory for the insightful discussions we had during the course of this research, as well as other faculty and staff in the Department of Computer Science and Engineering. Especially, I am very thankful to Dr. Abdol-Hossein Esfahanian, Dr. S. Masoud Sadjadi, Dr. Jonathan K. Shapiro, Dr. Chiping Tang, Dr. Zhinan Zhou, Dr. Peng Ge, Dave Knoester, Eric Kasten, Borzoo Bonakdarpour, and Linda Moore. In addition, I would like to thank several other wonderful people in the Michigan State University community who helped and inspired me in many ways. In particular, I thank Peter Briggs and Zeynep Altinsel at the Office of International Studies and Programs. Furthermore, I would like to thank all my teachers during my education. I would like to express thanks to Dr. Majid Azarakhsh, Dr. Mansour Z. Jahromi, Dr. Ahmad Towhidi, Dr. Farhang Vessal, Dr. Fariborz Sobhanmanesh, and Dr. Manijeh Keshtgari, at Shiraz University, where I did my undergraduate studies, who have always supported me. Last but not least, I would like to thank my family, and friends: my mom, dad, and brother who have always been encouraging and supportive; and friends—too many to name here—who have lightened my life.

¹This work has been supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and in part by National Science Foundation grants CCR-9912407, EIA-0000433, EIA-0130724, and ITR-0313142.

TABLE OF CONTENTS

LIST OF TABLES	ix
LIST OF FIGURES	x
1 Introduction	1
2 Background and Related Work	7
2.1 Adaptive Middleware	8
2.2 Cross-Layer Adaptation	13
2.3 Overlay Networks	18
2.4 Dynamic Service Composition	23
2.5 Need for Service Clouds	30
3 Service Clouds Infrastructure	32
3.1 Requirements and Challenges	32
3.2 System Model	36
3.3 Architecture	38
3.4 Prototype Implementation	42
4 Deep Service Clouds: Experimental Evaluation on the Internet	49
4.1 Experimental Testbed and Implementation	50
4.2 Case Study: TCP-Relay	52
4.2.1 Basic Operation	52
4.2.2 Implementation Details	55
4.2.3 Empirical Results and Analysis	61
4.2.4 Adaptive TCP-Relay	69
4.3 Case Study: Multipath Connections	78
4.3.1 Basic Operation	78
4.3.2 Implementation Details	80
4.3.3 Empirical Results and Analysis	82
4.4 Related Work	86
4.5 Summary	91
5 Mobile Service Clouds: Extending Service Clouds to the Wireless Edge	93
5.1 Mobile Service Clouds Model	95
5.2 Prototype Implementation	99
5.3 Case Study: High-Quality Wireless Video Streaming	101
5.3.1 Basic Operation and Experimental Setup	101
5.3.2 Implementation Details	104

5.3.3	Empirical Results and Analysis	107
5.4	Case Study: Reconfigurable Proxies	112
5.4.1	Basic Operation and Experimental Setup	112
5.4.2	Empirical Results and Analysis	115
5.5	Case Study: Seamless Mobility	116
5.5.1	Basic Operation and Experimental Setup	118
5.5.2	Empirical Results and Analysis	120
5.6	Summary	121
6	Dynamis: Dynamic Service Composition for Distributed Stream Processing	124
6.1	Problem Formulation	126
6.2	Service Composition Framework	129
6.3	Dynamis Algorithm	139
6.4	Experimental Evaluation	142
6.4.1	Test Setup and Procedure	142
6.4.2	Performance Analysis	144
6.5	Related Work	164
6.6	Summary	169
7	Conclusions and Future Directions	170
7.1	Contributions	171
7.2	Problem Revisited	172
7.3	Future Directions	180
	BIBLIOGRAPHY	184

LIST OF TABLES

4.1	Major public methods of the <code>RttMonitor</code> class.	59
4.2	List of nodes in the TCP-Relay experiment.	62
4.3	List of nodes in the adaptive TCP-Relay experiment.	74
4.4	List of nodes in the MCON experiment.	84
5.1	Video specifications in the high-quality wireless streaming experiment. . . .	104
5.2	Overall packet loss and quality in the wireless video streaming experiment.	109
5.3	List of nodes in the experiment of self-management at the wireless edge. . .	113
5.4	List of nodes in the seamless mobility experiment.	120
6.1	Service path quality metric formula notations	134
6.2	Dynamis algorithm parameters.	139
6.3	Configuration of parameters in the Dynamis experiment.	144
6.4	List of nodes in the Dynamis experiment.	145

LIST OF FIGURES

1.1	Conceptual view of the Service Clouds infrastructure.	4
2.1	The Adapt architecture.	10
2.2	DynamicTAO framework.	11
2.3	ACT configuration in a simple CORBA application.	13
2.4	Cross-layer adaptation.	14
2.5	Odyssey architecture.	16
2.6	Architecture of quality-channel in DEOS.	17
2.7	GRACE-1 cross-layer adaptation framework.	18
2.8	Overlay network example.	19
2.9	The iOverlay architecture.	20
2.10	The CANS organization.	22
2.11	Dynamic service composition.	24
2.12	Overall GridKit architecture.	25
2.13	The GridKit Component Framework Model.	26
2.14	SpiderNet: service composition system architecture.	29
2.15	DSMI architecture layers.	30
3.1	Vertical and horizontal cooperation.	37
3.2	Relationship of Service Clouds to other system layers.	39
3.3	The Service Clouds architecture.	40

3.4	The Service Clouds prototype.	44
3.5	Snippet of a sample XML inquiry packet.	46
4.1	Map of PlanetLab nodes used in the Deep Service Clouds experiments. . .	50
4.2	The Deep Service Clouds prototype.	53
4.3	An example of a TCP relay.	54
4.4	An example of TCP-Relay run-time operation.	58
4.5	Snippet of the TCP-Relay service class.	60
4.6	TCP-Relay overall empirical results on PlanetLab.	65
4.7	TCP-Relay results for a selected pair.	67
4.8	Internet path of the TCP-Relay sample.	68
4.9	Basic flowchart of the adaptive TCP-Relay algorithm.	70
4.10	Flowchart of the implemented adaptive TCP-Relay algorithm.	72
4.11	Adaptive TCP-Relay evaluation: data transfer times for 128MB	75
4.12	Adaptive TCP-Relay evaluation: performance ratios	77
4.13	Basic operation of MCON.	79
4.14	MCON inquiry packet processing within Deep Service Clouds.	83
4.15	MCON empirical results on PlanetLab (average loss rate and robustness). .	87
4.15	MCON empirical results on PlanetLab (dynamics and CDF of loss rate). . .	88
4.15	MCON empirical results on PlanetLab (average and CDF of delay).	89
5.1	Example scenario involving Mobile Service Clouds.	96
5.2	Instantiation of the prototype for Mobile Service Clouds.	100
5.3	The experimental testbed for high-quality wireless video streaming.	102

5.4	MetaSendMSocket architecture.	105
5.5	TRAP/J operation at compile time.	106
5.6	Software setup on the experimental testbed.	108
5.7	Percentage of packets received in the wireless video streaming experiment.	109
5.8	Video snapshots.	110
5.9	Video quality measurement in the wireless video streaming experiment.	111
5.10	The experimental setup for self-managing services at the wireless edge.	114
5.11	Packet loss during proxy failure at the wireless edge.	117
5.12	The experimental setup for seamless mobility.	119
5.13	Audio packet loss rate in the seamless mobility experiment.	122
6.1	Service graph example (highlighting a service path).	127
6.2	Examples of service graphs.	128
6.3	Service ordering constraints and corresponding service paths.	130
6.4	Overall service composition framework for autonomic communication.	132
6.5	Basic operation of the composition probing algorithm.	133
6.6	Example run of the Dynamis probing algorithm (steps 1 and 2 of 4).	137
6.6	Example run of the Dynamis probing algorithm (steps 3 and 4 of 4).	138
6.7	Dynamis algorithm pseudocode.	140
6.8	Quality and delay of service paths vs. overhead of probing.	147
6.8	Quality and delay of service paths vs. overhead of probing (continued).	148
6.9	Effect of T on the service path delay and probing overhead.	150
6.10	Effect of Q on the service path delay and probing overhead.	151

6.11	Quality and end-to-end delay of service paths as load increases.	152
6.12	Average end-to-end delay of service paths as load increases (pairs 1 and 2).	154
6.12	Average end-to-end delay of service paths as load increases (pairs 3 and 4).	155
6.12	Average end-to-end delay of service paths as load increases (pairs 5 and 6).	156
6.12	Average end-to-end delay of service paths as load increases (pairs 7 and 8).	157
6.12	Average end-to-end delay of service paths as load increases (pairs 9 and 10).	158
6.13	Sample of end-to-end service path delay as load increases (pairs 1 and 2).	159
6.13	Sample of end-to-end service path delay as load increases (pairs 3 and 4).	160
6.13	Sample of end-to-end service path delay as load increases (pairs 5 and 6).	161
6.13	Sample of end-to-end service path delay as load increases (pairs 7 and 8).	162
6.13	Sample of end-to-end service path delay as load increases (pairs 9 and 10).	163
6.14	Sample load distribution on nodes after service path setup for all pairs.	165
7.1	Illustration for adaptation scenarios.	175
7.2	Basic model for self-managing services.	176
7.3	MetaRelay architecture.	177
7.4	Multi-path connection relay modeled as MetaRelay.	178
7.5	Wireless link error-correction modeled as MetaRelays.	179
7.6	Seamless mobility case study modeled using MetaRelays.	181
7.7	Ecosystem stream processing example.	183

Chapter 1

Introduction

Computer applications play an increasing role in managing their own execution environment. This trend is due in part to the emergence of autonomic computing technologies [1, 2], where systems are designed to respond dynamically to changes in the environment with only limited human guidance. Autonomic systems require adaptation to balance several conflicting and possibly crosscutting concerns, which include quality of service, resource consumption, and changing security policies [3]. Further, autonomic computing supports pervasive computing [4], which removes the traditional boundaries for how, when, and where humans and computers interact. As computing is woven into the fabric of daily life, regardless of physical location, autonomic computing is needed to ensure that the constituent services continue to be provided, despite dynamic conditions and component failures. One area in which autonomic behavior can be particularly helpful is communication-related software; autonomic computing can be used to improve quality of service, support fault tolerance, and enhance security in the presence of dynamic network conditions.

Autonomic communication services promise to enhance quality of communication by composing and adapting services at run time in response to changing user needs and varying network conditions. Examples of enhanced communication qualities include secure connections, robust multimedia streaming, high-speed bulk data transfer, stream adaptation to overcome resource restrictions of mobile clients, and seamless support for user mobility. However, coordinating the actions of multiple software components, distributed across multiple platforms, is a nontrivial task. An infrastructure to support such communication services requires both cross-layer (vertical) and cross-platform (horizontal) cooperation. Designing such a system is a challenging task is complicated by the nature of adaptive software, uncertainty in the execution environments, and heterogeneity among software modules and hardware platforms. Moreover, a service infrastructure itself must be resilient to malfunctions and should not, due to its complexity, make the communication more fragile.

We argue that a generic, extensible infrastructure for supporting such interactions can be very useful to the development and deployment of autonomic communication services. Typically, autonomic service composition comprises *algorithms* to probe a network and select distributed service entities, *protocols* to configure and bind chosen entities to form a service graph, and *adaptation* of distributed entities as conditions change and failures occur. Identifying building blocks for an autonomic infrastructure and defining their interactions is not possible without extensive study of related areas, construction of prototypes, and experimental evaluation in real world environments.

Many approaches to software adaptation are based on *adaptive middleware* [5–11], which enables distributed software to detect and respond to changing conditions in a man-

ner orthogonal to, and in many cases independent of, the business logic of the application. Research in this area over the past several years has been extensive and has provided a better understanding of key concepts relevant to autonomic computing. Moreover, *cross-layer* cooperation mechanisms [12–17] enable adaptation at multiple layers of a system in a coordinated manner that is optimal for the system as a whole, and in ways not possible within a single layer. In *overlay networks* [18], in which end hosts form a virtual network atop a physical network, the presence of *hosts* along the paths between nodes enables intermediate processing of data streams, without modifying the underlying network protocols or router software. Recently, researchers have investigated ways to use overlay networks to support dynamically configurable communication services [19–24]; several contributions are discussed in Chapter 2. While the above techniques can be used to simplify development of adaptive services, an integrated framework is needed that combines these techniques to support the design and rapid deployment of different services.

This dissertation investigates a model for such an infrastructure, called *Service Clouds*, that supports dynamic instantiation and reconfiguration of autonomic communication services. In the Service Clouds model, collaborating nodes form an overlay network that supports cross-platform adaptation and use cross-layer mechanisms locally. The key insight is that the overlay nodes provide a “*blank computational canvas*” on which services can be instantiated as needed, and later reconfigured in response to changing conditions. Figure 1.1 shows a conceptual view of Service Clouds. Individual nodes in the clouds of hosts use adaptive middleware and cross-layer collaboration to support autonomic behavior; an overlay network among these nodes serves as a vehicle to support cross-platform adaptation. The Service Clouds infrastructure is designed to be extensible: a suite of low-level

services for local and remote interactions can be used to construct higher-level autonomic services. We implement a prototype of the Service Clouds model and evaluate it experimentally. We show how the Service Clouds infrastructure provides high-performance communication atop the wired Internet by instantiating distributed service entities dynamically, and combining cross-layer, cross-platform cooperation to compose adaptive services. To support pervasive mobile computing, we extend the design to the wireless edge of the Internet, where collections of nodes close to the wireless edge can dynamically instantiate, configure, and migrate proxies to support robust streaming for mobile clients.

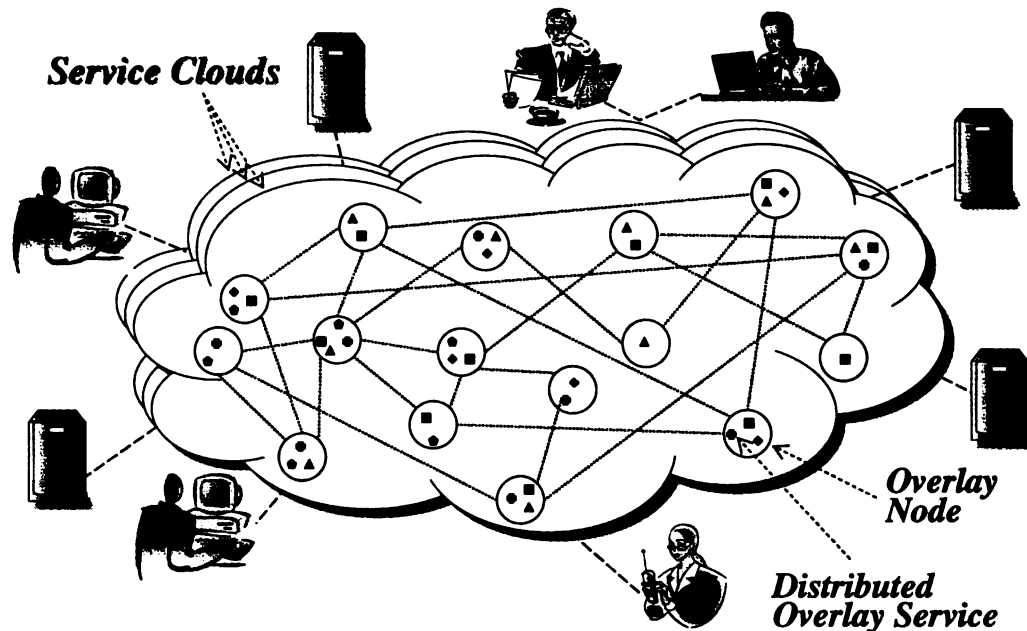


Figure 1.1: Conceptual view of the Service Clouds infrastructure.

Thesis Statement. *The Service Clouds model and corresponding infrastructure provide an effective means to construct and maintain autonomic communication services in both the Internet and mobile computing environments.*

This dissertation makes the following contributions.

1. We design and evaluate *Service Clouds*, an overlay-based model and architecture [25]. We show how to organize constituent components and define interactions among building blocks. We implement a prototype of Service Clouds based on the proposed architecture, which combines overlay algorithms into an adaptive middle-ware layer. This approach yields a design guideline and toolkit, as well as an integrated framework, to rapidly prototype enhanced communication services.
2. We evaluate the effectiveness of Service Clouds to support autonomic communication on the wired Internet [26]. We refer to this model as *Deep Service Clouds* and use the prototype to deploy two specific services: *TCP-Relay*, in which a node is selected and configured dynamically to serve as a relay to expedite bulk data transfer; and *MCON*, a multipath connection algorithm that exploits network topology to dynamically find and establish a high-quality secondary path between two nodes, as a shadow connection to the primary path to support high-quality streaming. In addition to demonstrating the usefulness of the Service Clouds architecture in designing and rapidly implementing these services, we experimentally evaluate the effectiveness of the infrastructure to achieve high performance on an Internet testbed.
3. We investigate the extension of the Service Clouds design to the wireless edge of the Internet. This model, called *Mobile Service Clouds*, supports pervasive mobile computing under highly dynamic conditions and frequent relocation of users [27]. Specifically, collections of overlay hosts dynamically deploy services close to the wireless edge to accommodate needs of mobile clients. We extend the prototype to address multimedia streaming, and demonstrate how dynamic creation, configura-

tion, and migration of proxies support QoS streaming and mobility [25, 28].

4. We address dynamic composition of distributed services atop an overlay network. We introduce and evaluate *Dynamis*, a generic algorithmic approach to locating suitable nodes on which to map service entities with minimal probing overhead while finding high-quality service paths [29]. Empirical results, from experiments using the Service Clouds prototype, show this approach can dramatically reduce probing traffic overhead, when applied to probing methods, while producing high-quality mapping of services to nodes.

The remainder of this dissertation is organized as follows. Chapter 2 provides background and related work on adaptive middleware, cross-layer adaptation, overlay networks, and adaptive communication services. Chapter 3 presents the Service Clouds model; we discuss design challenges, describe the current architecture, and explain building of prototype software. Chapter 4 investigates deployment of services on the wired Internet (Deep Service Clouds) and evaluates the design and implementation in case studies for high-performance data transfer and resilient communication. Chapter 5 explores the Mobile Service Clouds operation and presents case studies for robust, high-quality streaming to mobile clients. Chapter 6 introduces *Dynamis*, a generic algorithmic approach for dynamic service composition that facilitates high-quality mapping of distributed services onto an overlay network with low traffic overhead. Finally, in Chapter 7, we summarize the contributions of this study, revisit the problem and consider higher-level abstractions based on lessons learned, and discuss future research directions.

Chapter 2

Background and Related Work

In this chapter, we review the main areas of research supporting autonomic communication services. Realizing self-adaptive behavior typically requires cooperation of multiple software components. This cooperation may be *vertical*, involving different system layers, or *horizontal*, involving software on different platforms. Many approaches to vertical cooperation are based on adaptive middleware, since middleware is an ideal location to implement and manage self-monitoring and adaptive behavior. Many approaches to horizontal cooperation involve the use of overlay networks, in which end hosts form a virtual network atop a physical network and provide an adaptable and responsive chassis for communication. First, we discuss adaptive middleware and key concepts that support dynamic adaptation. Second, we review cross-layer adaptation and identify supporting techniques essential to it. Third, we discuss the use of overlay networks as a chassis to support distributed communication services. Fourth, we describe and provide background on adaptive communication services. Finally, we address open areas in the design and operation of autonomic communication and motivate the need for our investigation.

2.1 Adaptive Middleware

The first major aspect of supporting autonomic communication services is software adaptation. Middleware is an intermediate layer between the application layer and the operating system layer that provides services to the applications. Traditionally, the main goal of middleware is to support distributed applications in heterogeneous computing environments. Middleware is a suitable place for adaptation because it can adapt services transparently to applications. An adaptive middleware layer facilitates dynamic reconfiguration, that is, changing the behavior or the operational parameters of middleware components, as well as inserting and removing components, as the application executes.

A large number of adaptive middleware projects have been conducted in the last few years [5, 7, 9, 30–36]. The contributions are far too numerous to review exhaustively here (see [3] for a survey). However, these works have produced or highlighted several key concepts that are important to adaptive software [37]. First, an *open* approach to middleware implementation [5], as opposed to the traditional “black box” approach, is important not only for supporting dynamic adaptation, but also for enabling collaborative adaptation with other layers and distributed platforms. Second, in *component-based design*, well-defined interfaces enable independent development and extension of components as well as dynamic reconfiguration and recomposition of services [38, 39]. Third, *reflection* [40] enables a program to inspect its own status and change its behavior during execution by inserting or removing components and changing their parameters. The use of reflection to guide adaptive behavior in middleware [7, 41] provides a principled approach that facilitates both code generation and formal verification. Fourth, a wide variety of methods

have been proposed to support middleware adaptation through *interception* and redirection of interaction among components [10, 11, 42, 43]; these techniques support transparent adaptation of interaction among layers and across different systems. Fifth, *separation of concerns* [44] enables separate development of the functional behavior (business logic) and the crosscutting concerns (e.g., quality of service, fault tolerance, and security) of an application. Software techniques such as aspect-oriented programming [45], which realize this separation, simplify software development, maintenance, and extension while promoting reusability. Finally, several projects have investigated *contract* mechanisms [8], which can be used to specify in a declarative manner the goals of the system and the possible adaptive actions.

In the remainder of this section, we review four middleware projects: Adapt, dynamicTAO, ACT, and QuO. These projects illustrate key concepts and techniques in adaptive middleware, which can be applied to deploy adaptation in autonomic communication services.

Adapt. In the Adapt project [5], Blair et al. have considered *openness* to support adaptive functionality needed by the applications. Figure 2.1 shows the Adapt architecture, which is based on CORBA [46] and models middleware as an object graph. In the object graph model, nodes of the graph represent “media processing” objects or “binding” components in middleware, and arcs are “local bindings” between the components. The control interfaces allow run-time inspection and adaptation of components. The entire object graph is dynamically reconfigurable and supports the concept of *open binding*, referring to dynamic adaptability of a compositional framework in a distributed environment.

For instance, Figure 2.1(b) depicts a binding configuration for pervasive video commu-

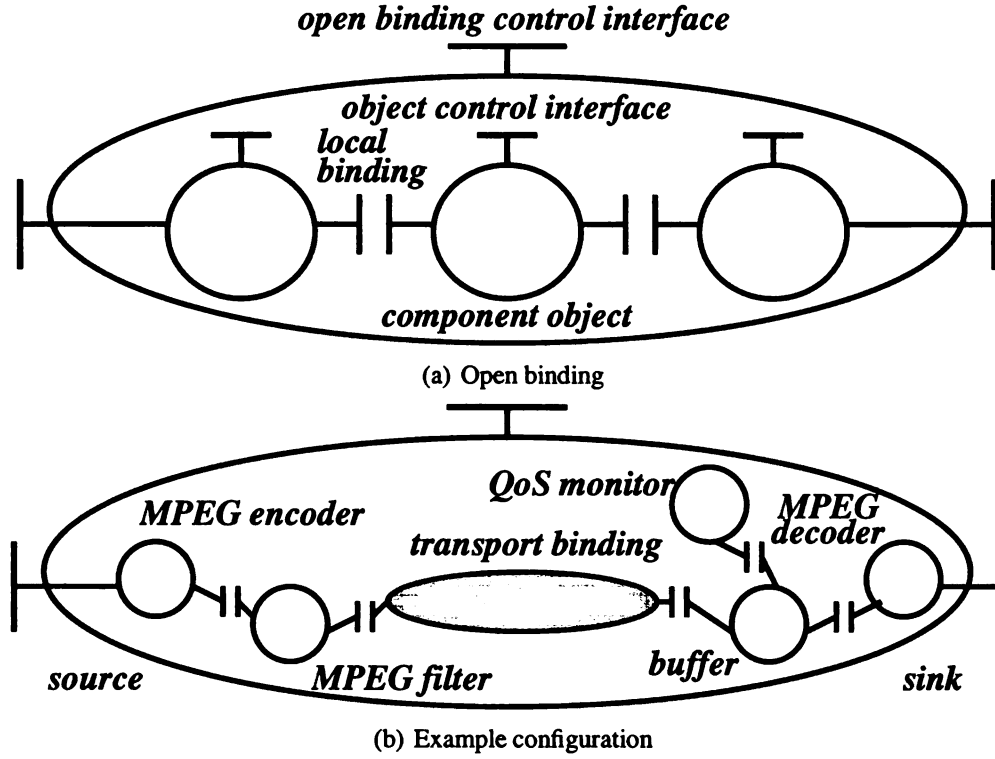


Figure 2.1: The Adapt architecture [5].

nication. When the mobile platform moves from a wired network to a wireless network, the QoS monitor detects an increase in jitter and notifies the video player application. In order to handle the jitter, the application invokes a control interface to adapt buffer parameters at run time. Further, by dynamically inserting and reconfiguring the MPEG filter and replacing a suitable transport binding, the application can adapt to various wireless environments [5]. The designers of Adapt also proposed a design model based on the reflection [47] and used it in the design of Open ORB [9, 48].

DynamicTAO. Kon et al. [7] introduced DynamicTAO, a *reflective* middleware platform based on CORBA [46]. DynamicTAO extends TAO [49], an open source reconfigurable middleware that supports only static reconfiguration, to support run-time reconfiguration. Figure 2.2 illustrates the dynamicTAO framework. DynamicTAO facilitates in-

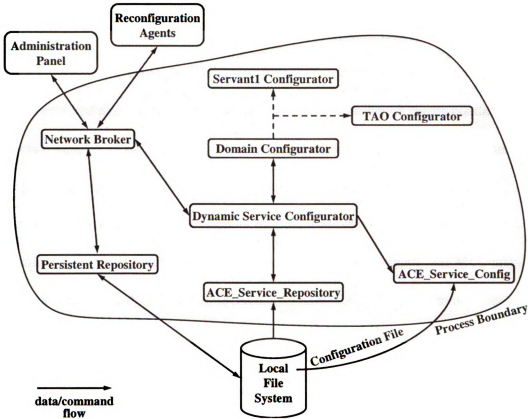


Figure 2.2: DynamicTAO framework [7].

spection and dynamic reconfiguration through component *configurators*. Component configurators manage dependency among components and their status to facilitate consistent run-time reconfiguration. The *Persistent Repository* stores dynamically loadable components. The *Network Broker* handles inspection and reconfiguration requests. The *Dynamic Service Configurator* deals with dynamic reconfiguration operations.

Using the experience gained in the dynamicTAO project, researchers developed the LegORB dynamic middleware [50]. LegORB has a tiny memory footprint that makes it suitable for embedded and mobile computing. DynamicTAO and LegORB have been employed in 2K [51], an adaptable network-centric OS that integrates middleware into the operating system within a *component-based* framework.

QuO. Zinky et al. [8] developed QuO, which uses *contracts* to support QoS adaptation, with decision making based on the current system status and the desired QoS. QuO extends CORBA [46] with a QoS description language (QDL). Developers can use QDL to define required adaptations for delivering expected QoS under changing conditions. The QuO code generators parse the IDL (CORBA's interface description language) and QDL descriptions and generate codes that are integrated into the QuO framework.

ACT. Interception enables developers to extend middleware transparently to the applications and without changing its infrastructure [52–54]. Sadjadi and McKinley [43] developed Adaptive CORBA Template (ACT), an interception-based approach for enhancing adaptive middleware that supports unanticipated adaptation and inter-operation among incompatible CORBA-based frameworks. The ACT framework uses a generic interceptor that intercepts requests, replies, and exceptions on the server and client sides. The intercepted communication is redirected to the ACT core, an adaptive framework that supports dynamic adaptation of the intercepted services with regard to functionality or quality. Using ACT, developers can add new functionalities to the existing CORBA-based applications, or enable legacy applications to benefit from CORBA-based middleware frameworks, such as QuO [8], at run time and without modifying application source code.

Figure 2.3 shows ACT configuration in the context of a simple CORBA application. The ACT framework comprises two main building blocks: a *generic interceptor* and an *ACT core*. A generic interceptor is registered with the ORB and intercepts all interactions (request, reply, and exception) between a client and a server. An intercepted interaction is forwarded to the ACT core, where it is dynamically adapted, for example, by changing frame rate of a reply stream when a mobile node connects to a low-bandwidth channel.

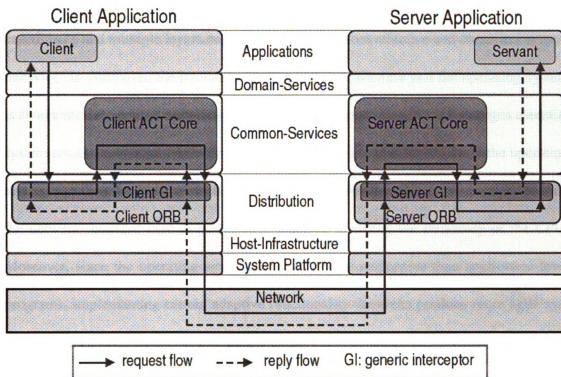


Figure 2.3: ACT configuration in a simple CORBA application [43].

2.2 Cross-Layer Adaptation

The second major aspect of supporting autonomic communication services is cross-layer adaptation, where adaptation takes place across multiple layers of a system in a coordinated manner. Vertical cooperation among different system layers is required to support self-adaptive behavior, which involves adaptation in multiple components of a system. Traditionally, since the role of middleware is to hide resource distribution and heterogeneity from the business logic of applications, middleware is a logical place to put adaptive behavior related to concerns such as QoS, security, and energy management. However, middleware alone cannot handle all types of adaptation. In general, adaptation can take place in different layers of a system: *application, middleware, operating system, and hardware*. As

shown in Figure 2.4, in a *cross-layer adaptation paradigm*, layer-specific adaptations are coordinated and multiple layers cooperate to achieve more effective and consistent system behavior. In particular, the interaction between the middleware and the operating system is an essential element in cross-layer cooperation. The operating system manages essential system resources that are typically not controllable from upper layers. Only the operating system can directly control adaptive behavior of hardware components, such as placing the network interface card in power save mode or adjusting the frequency of the CPU. Moreover, since the operating system is usually more responsive than application-level programs, implementing certain adaptive functionality there can produce more agile system behavior [55].

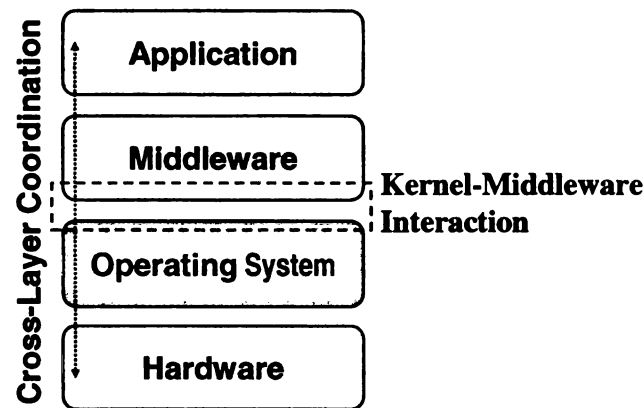


Figure 2.4: Cross-layer adaptation.

Several cross-layer adaptation frameworks have been proposed recently [12–15, 56, 57]. In these systems, adaptive entities at different layers cooperate to realize system-wide adaptive behavior. For example, middleware can make better decisions by working with the operating system, which has a system-wide view of resource usage. Conversely, the operating system can incorporate application-specific semantics into tasks such as filtering

packets and enforcing security policies. Such cooperation is facilitated by the realization of several key concepts [58]. First, in a similar way that openness in middleware supports application-specific adaptation, *openness* at any system layer supports flexibility and manageability required in a cross-layer model [55]. Second, *interception* of control or data flow is a basic mechanism for adaptation at any layer that is transparent to other layers to enhance qualities such as monitoring [59], security (e.g., by encryption/decryption in middleware or the operating system [54, 60]), and network failure resiliency (e.g., by re-routing at the network layer). Third, *modular design* supports cross-layer adaptation by enabling dynamic reconfiguration of modules at different layers and providing hooks to plug in interceptor modules that extend system (e.g., loadable kernel modules that wrap application interaction with the operating system to enforce access right policies [61]). Fourth, *contracts* facilitate negotiation between components in different layers (e.g., a QoS contract between middleware and the operating system enables the middleware to choose the right video encoder based on the availability of resources and MAC layer configuration [62]). Finally, *coordination* is a major aspect of cross-layer adaptation, as multiple layers need to cooperate to achieve consistent system behavior [13, 63]. Let us next consider three representative projects.

Odyssey. Satyanarayanan et al. [12, 64] developed Odyssey, an early application-aware adaptation framework in which applications specify their resource requirements to the operating system, and the operating system informs applications to adapt themselves whenever the expectations cannot be fulfilled due to limitation of resources. To realize this, Odyssey incorporates *interception* of application system calls at the operating system kernel. Figure 2.5 depicts the architecture of Odyssey. In this model, an *interceptor* controls access to

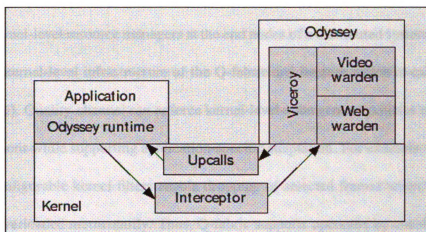


Figure 2.5: Odyssey architecture [65].

system resources by the applications registered with Odyssey. The interceptor intercepts file system calls and redirects them to a central manager called *Viceroy*. *Viceroy* monitors resources and notifies applications of resource limitations. Fetching of data is managed by the *wardens* that facilitate data communication between client and servers. They contain meta-data information that specifies resource requirements for data sets with different characteristics, for example, the meta-data can specify bandwidth and energy consumptions for streams of the same video with various color-depths. Whenever a resource requirement cannot be fulfilled, *Viceroy* sends a notification *upcall* to the application. When the application receives a notification, it adapts itself by interacting with the corresponding warden to choose a data set that satisfies resource expectations.

DEOS. An example of a cross-layer framework that focuses on application and operating system cooperation is the *Q-fabric* architecture framework in the DEOS project [66] by Schwan et al. The DEOS project addresses dynamic QoS when availability of system resources is constantly changing, for example, in distributed multimedia applications where

CPU and network bandwidth vary continuously [55]. Q-fabric consists of application adaptors and kernel-level resource managers at the end nodes of a distributed system. Figure 2.6 shows the kernel-level infrastructure of the Q-fabric architecture which is called *quality-channel* [14]. Quality-channel can enforce kernel-level management policies transparently to applications while supporting application-specific adaptation. For example, application-specific configurable kernel filters enable dropping of selected frames when the network becomes overloaded momentarily. Thus, Q-fabric supports openness by enabling application participation in the kernel-level adaptation.

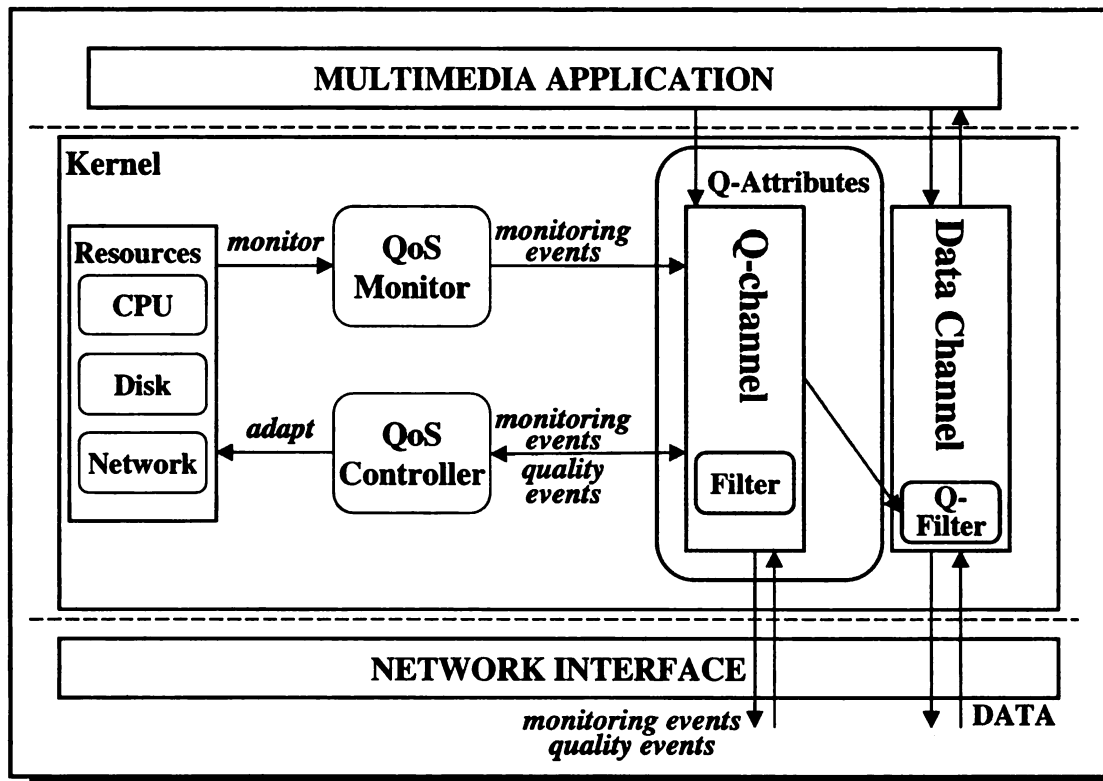


Figure 2.6: Architecture of quality-channel in DEOS [55].

GRACE. Resource access and consumption at each layer affect the entire system and need to be *coordinated* and controlled in a cross-layer manner. The GRACE project [13],

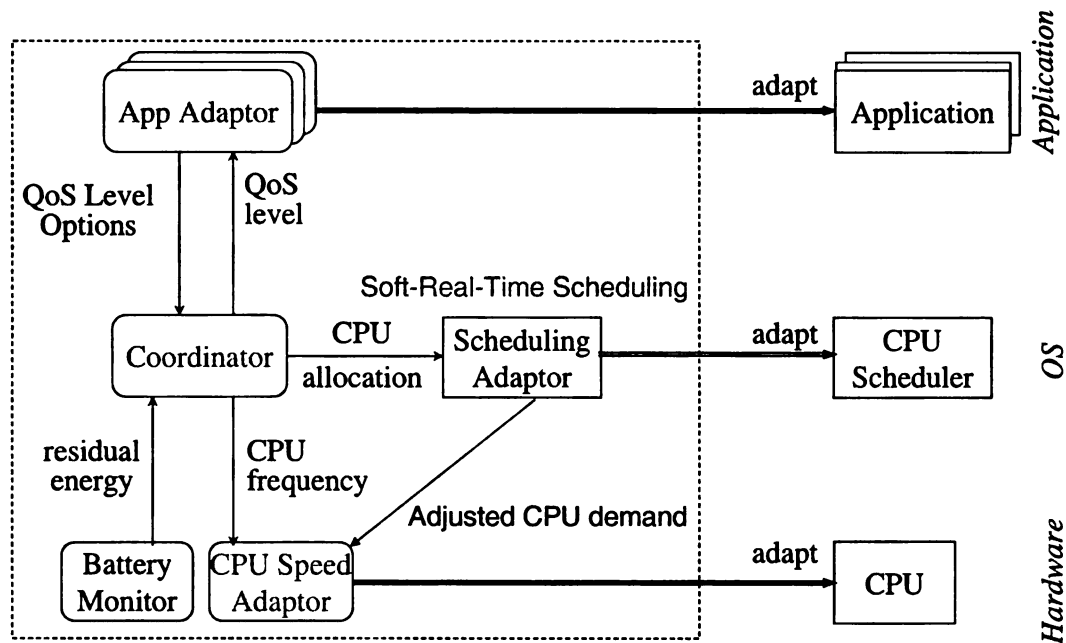


Figure 2.7: GRACE-1 cross-layer adaptation framework [57].

by Adve et al., uses a cross-layer approach to support optimized adaptation in multimedia applications within the constraints of energy, time, and bandwidth. The key theme in the GRACE model is that local adaptations at different system layers cooperate to accomplish an optimal global adaptation. The GRACE architecture includes hardware, operating system, network, and application layers. To achieve maximum productivity, a resource coordinator interacts with different layers of the system to manage dynamic global adaptation, as shown in Figure 2.7

2.3 Overlay Networks

The third major aspect of supporting autonomic communication services is overlay networks, in which end hosts form a virtual network atop a physical network. Overlay net-

works provide a substrate for horizontal cooperation and have been used to support a variety of services, including end-system multicast [67, 68], structured peer-to-peer systems [69–71], and resilient routing [18]. Figure 2.8 shows a simple overlay network. If the physical link $F - G$ fails, the direct communication between nodes A and C breaks down. However, using overlay routing, those two nodes can continue communication via node D . Because they execute on hosts, overlay networks can provide an adaptable and responsive chassis on which to implement communication services for many distributed applications [21–24, 72–75]. In this section, we review representative projects that address three major subjects in constructing overlay-based frameworks, including routing and message dissemination, data adaptation services, and implementation, respectively.

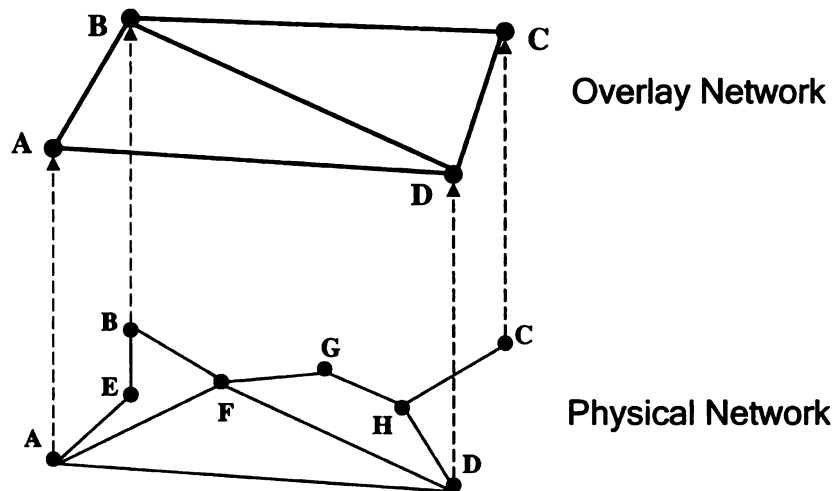


Figure 2.8: Overlay network example.

iOverlay. Li et al. [21] developed iOverlay, a lightweight middleware that facilitates construction of overlay applications by providing a *message switching engine* and primitives for monitoring and reporting on system status (e.g., node/link failures, link delay, and channel throughput). iOverlay addresses high-performance communication and implements its message switching engine from scratch in C++ with simple well-defined in-

terfaces. Effectively, iOverlay provides a low-level middleware layer for message-driven overlay applications.

Figure 2.9 shows the iOverlay architecture. The architecture considers three layers in a distributed overlay system: (1) the *engine*, which switches messages; (2) the *algorithm*, which implements an overlay protocol; and (3) the *application*, which interprets the data carried in messages at the endpoints. The iOverlay facilitates deployment of all these layers by providing: a high-performance general message switching engine; common elements used in overlay algorithms; typical applications to run atop an algorithm; and a graphical utility, called *observer*, to monitor and debug a distributed overlay system.

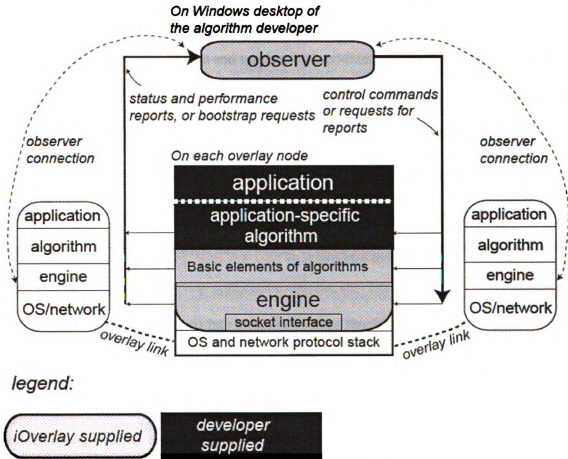


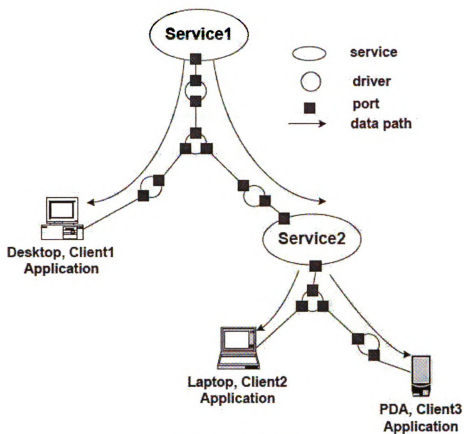
Figure 2.9: The iOverlay architecture [21].

CANS. Fu et al. [24] introduced CANS, an infrastructure for *dynamic creation and*

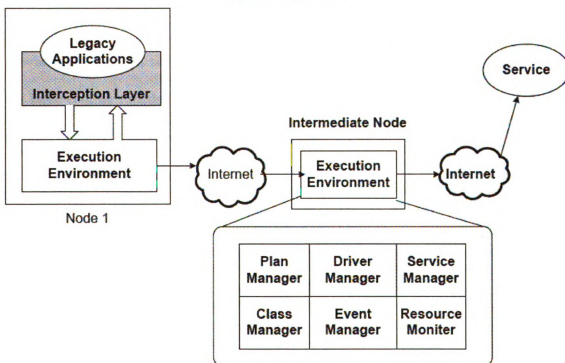
reconfiguration of services along a data path, with a primary focus on connections between the Internet hosts and mobile clients. Since mobile devices have limited resources, data may need to be suitably adapted as it approaches the wireless edge of the network. The CANS infrastructure composes a service path that adapts a data stream by inserting, removing, and reconfiguring components at intermediate nodes and end nodes. To deploy path services, CANS provides three main mechanisms: high-level type-based component and resource specification, automatic path service creation according to performance preferences (desired QoS), and low-overhead dynamic reconfiguration of path services.

Figure 2.10 shows the CANS view of a network. In this view, Fig. 2.10(a), basic organization of a network consists of *application*, *services*, and *data paths*. The CANS model enables adaptation of data and services by introducing *Execution Environment (EE)*, Fig. 2.10(b). The EE run-time environment supports downloading components, composing a complex service along a data path, and reconfiguring services. The EE provides an event manager to support communication among service components and a unit to monitor system conditions. CANS supports two types of services: *intermediate services*, and *drivers*. Intermediate services are extant heavyweight entities such as a web cache. Drivers are lightweight entities that provide application-specific services, such as a video transcoder, and are injected by applications into a data path when required.

MACEDON. Rodriguez et al. [22] proposed MACEDON, a language-based methodology to *automatically generate code* for an overlay service implementation. It facilitates describing an overlay algorithm and automatically generating code from the description, thus, relieving algorithm designers and developers from implementation details. Developers can specify behavior of overlay algorithms as event-driven finite state machines (FSM).



(a) Basic organization



(b) Execution environment

Figure 2.10: The CANS organization [24].

The FSM comprises *states* (e.g., a node in joining state), *events* (e.g., message reception), and *transitions* triggering actions in response to events (e.g., transmitting a message, updating a table entry). The MACDON framework provides a C++ like language to define an algorithm behavior and generates code that runs on platforms such as the PlanetLab Internet testbed and the ModelNet emulator. Moreover, an algorithm can specify a base overlay algorithm and execute on top it. This approach supports reusability of performance-tuned code as well as evaluation of different configurations (e.g., by changing an underlying distributed hash table). Also, generation of code using the same engine and library functions increases consistency in comparing different approaches together.

2.4 Dynamic Service Composition

The fourth major aspect of supporting autonomic communication services is dynamic composition of service elements distributed in an overlay network. In a service overlay, nodes provide data services, such as video transcoding and protocol conversion, in addition to the basic routing functionality. It is a challenging problem to satisfy QoS expectations through dynamic composition and adaptation of service elements distributed over multiple nodes. With the emergence of overlay networks and services on the Internet, the service composition problem has attracted many researchers [19, 23, 76–83].

Service composition in its simplest form consists of connecting a number of service units together to provide an end-user service. For example, when a mobile user travels outside of the service provider area, a service composition strategy that stretches over two service providers can maintain user access by enabling authentication, billing, and required

communication routing. Deployment of services in an adaptive middleware framework supports dynamic configuration of such complex services distributed in an overlay network.

Dynamic service composition involves five basic stages shown in Figure 2.11. First, the system has to *discover* available services. Second, the system has to *select* suitable services that satisfy requested QoS (qualified services). Third, the system has to *compose* a primary service path, and possibly backup service paths. It inspects the collection of qualified services and identifies a set of services that can together provide the requested complex service (chosen service paths). It then notifies overlay nodes along the path to reserve required resources. Fourth, the system has to *configure* each service on an overlay node according to the required functionality. Finally, the system needs to automatically *adapt* composed services at execution time in response to changes in the communication and computing environment.

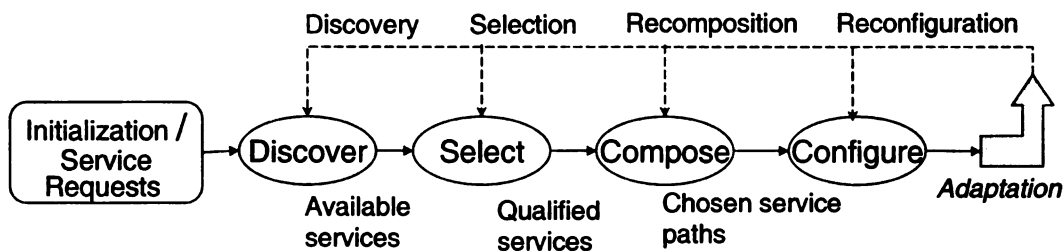


Figure 2.11: Dynamic service composition.

As shown in the figure, adaptation can take the system back to any of the aforementioned basic stages. *Reconfiguration* adapts an individual service, for example, plugging in a new video encoder at a video proxy when a user switches from wired to wireless connection. *Recomposition* sets up a new service path by inserting or removing services. For example, adding a new encryption service in a data stream path, or switching to a backup service path when the current path fails. *Selection* involves evaluating discovered services

and updating the pool of qualified services. Finally, returning to the *discovery* stage takes the system back to the beginning of the service composition procedure. The further the system goes back in the service composition procedure, the more overhead and time is taken to adapt the system. Below, we review three representative projects and describe their approaches in support of service composition and adaptation.

GridKit. Coulson et al. [77] developed GridKit, a communication infrastructure that supports an extensible set of “interaction types.” Typical interaction types in service-oriented architectures include message-passing and request-reply. GridKit enhances communication services to support interaction types such as publish-subscribe, QoS media-streaming, and tuple-space. The basic theme to support extensible interaction types is based on the concept of *pluggable overlay networks*. GridKit supports pluggable and reconfigurable overlay networks by implementing them based on the OpenCOM [84] reflective *component model (CF)*. By combining a reflective component framework and overlay networks, GridKit realizes a reconfigurable communication infrastructure, and at a higher level, supports four key middleware “domains” shown in Figure 2.12. Atop these domains the Grid Services layer supports operation within Grids by facilities such as Web Services API.

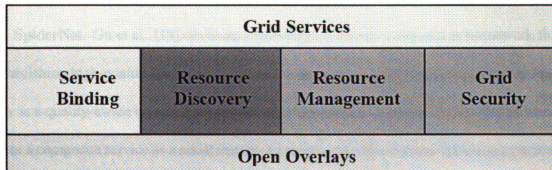


Figure 2.12: Overall GridKit architecture [77].

Figure 2.13 shows the GridKit CF, which is based on the OpenORB/OpenCOM model. A small memory footprint of 250KB makes it suitable for mobile computing. The internal structure is a graph of sub-components with XML-based architectural descriptions of valid compositions. The *IAccept* interface supports plugging of descriptions, which define valid compositions. Based on OpenCOM, the *ICFMetaArchitecture* is the reflective interface that allows dynamic inspection and manipulation of the internal structure. Reconfiguration of sub-components takes effect only if it conforms to valid composition descriptions.

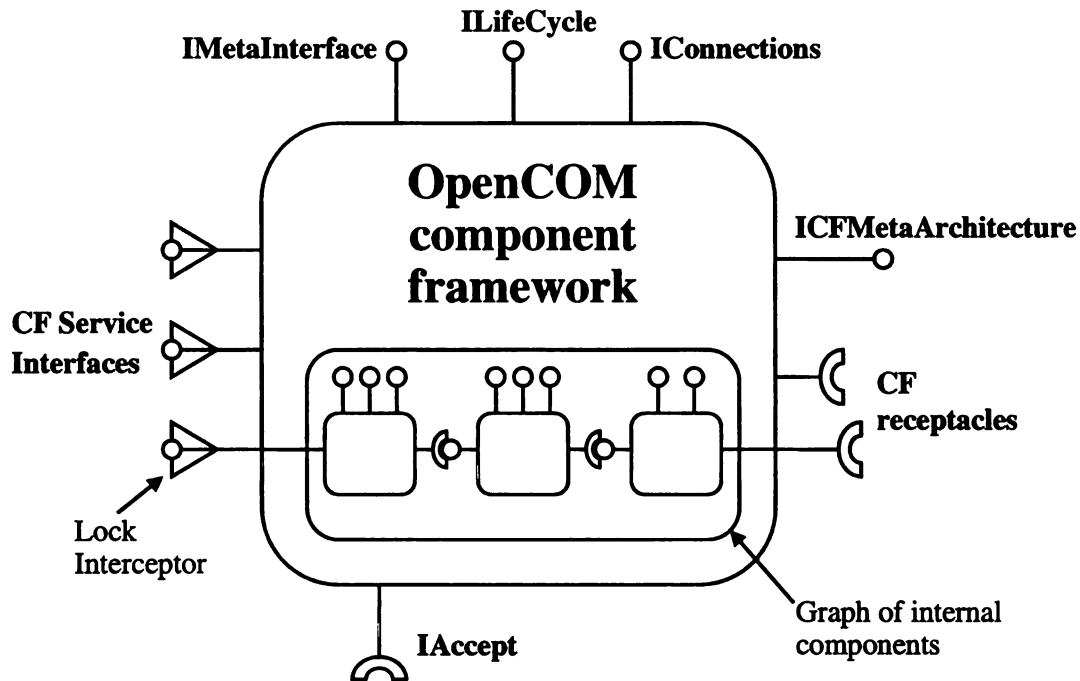


Figure 2.13: The GridKit Component Framework Model [20].

SpiderNet. Gu et al. [19] developed SpiderNet, a service composition framework for establishing high-quality fault-tolerant service paths in peer-to-peer systems. The key feature is a quality-aware bounded probing protocol for service composition. SpiderNet identifies a composite service as a set of distributed functions with specified QoS requirements. To compose a service, the probing mechanism finds a sequence of nodes that can provide

required functions with the specified quality (depending on the availability of resources). Then, the system composes a primary service path and maintains a number of possible service paths as backups. A proactive method for failure recovery, in soft real-time streaming, monitors the condition of backup paths with low-overhead probing and replaces a failed service path with a backup.

Figure 2.14 shows the SpiderNet view of service composition. A composite service consists of a set of functions forming a graph of functions along with corresponding QoS requirements. The basic idea is to probe *only* nodes capable of performing required functions. SpiderNet uses distributed hash table (DHT) to identify nodes capable of doing a task. The source node initiates probing by sending a probe to a set of its neighbor nodes capable of performing the first required task in the function graph. The probe contains the function graph, QoS requirements, and the set of nodes it has traversed along with the resources status of each. Upon receiving a probe, a node finds its neighbor nodes capable of performing the next task in the function graph, adds its resource status information to the probe, and forwards the probe only to those neighbors. Probing continues until a collection of probes, indicating possible service paths, reach the destination. The destination inspects probes, calculates the quality of each possible service path, and selects the service path with the lowest load. Then, it sends an acknowledgement message towards the source that also reserves required resources along the service path. Finally, the source node begins using the best selected service path and keeps the other selected service paths as backup when the primary path fails. A failure happens when one or more services on a service path cannot satisfy the specified task with the expected quality of service. In this case, SpiderNet chooses a backup path which does not contain the failed service but has the most

number of overlapping services with the failed path. This enables fast recovery since the system needs to setup a minimum number of services to form the new service path. Recently, the SpiderNet model has been used in a hybrid approach that combines distributed probing with coarse-grained global state management for optimal component composition in streaming applications [85].

DSMI. Kumar et al. [78] introduced DSMI, a scalable resource-aware approach to distributed stream management. Specifically, this approach addresses adaptive aggregation of streams at overlay nodes. An underlying layer performs dynamic network partitioning based on the availability of resources, producing a data-flow graph for efficient processing of the streams. DSMI uses a middleware framework to (re-)configure overlay service nodes and processing components according to the data-flow graph. At the application layer, DSMI supports high-level descriptions of application data flows and processing operations.

Figure 2.15 shows the DSMI architecture which has three main layers: application, middleware, and network partitioning. The *application layer* provides a SQL-like language to describe a stream along with its attributes. This description is compiled to produce a graph for the corresponding data-flow consisting of data-flow paths (graph edges) that connect processing nodes (graph vertices) together for distributed processing (such as aggregation and filtering) from sources towards sinks. The *middleware layer* consists of two components: a publish-subscribe and a resource monitoring service. This layer deploys the data-flow graph on the overlay network and reconfigures data-flow paths and processing according to the resource availability and specified QoS. The stream transformation components are written in a C-like language and are compiled on data processing nodes dynamically. The underlay *network partitioning layer* constructs and maintains a

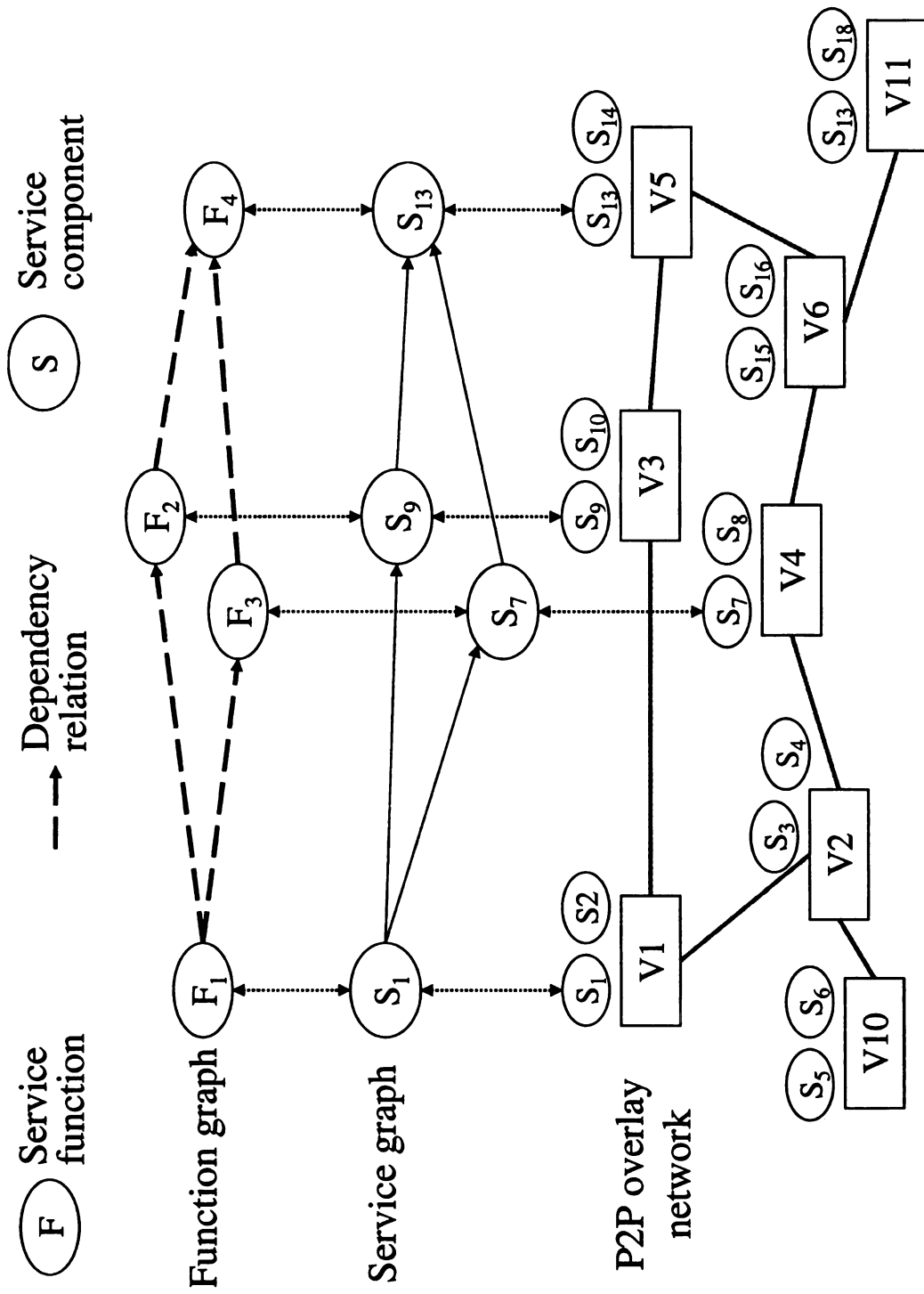


Figure 2.14: SpiderNet: service composition system architecture [19].

<i>Application Layer: Data-Flow Parser</i>	
<i>pub-sub Middleware ECho</i>	<i>Resource Monitoring PDS</i>
<i>Underlay Layer: Network Partitioning</i>	

Figure 2.15: DSMI architecture layers [78].

hierarchical organization of nodes that clusters them according to attributes such as end-to-end delay or bandwidth. Nodes in a cluster have global information about the cost of paths in that cluster, and a coordinating node from each cluster participates in the next level of the hierarchical structure. This multi-level structure provides an abstraction of the physical overlay network for the upper layers. DSMI provides an algorithm that maps the data-flow graph onto the physical overlay network such that the cost of distributed data-flow processing is minimized. As the overlay network conditions change, the system reacts by reconfiguring this mapping.

2.5 Need for Service Clouds

A general infrastructure for autonomic communication services needs to support and combine both vertical and horizontal interactions. Such an infrastructure can use adaptive middleware as an enabling technology. Projects such as Odyssey [64], DEOS [66], and GRACE [13] address cooperation among multiple system layers, which is necessary to achieve consistent adaptive behavior. However, these projects focus on adaptation either on a single node or at communication endpoints, rather than distributed cooperation among multiple nodes. On the other hand, overlay network projects such as RON [18] and iOver-

lay [21] address cooperation atop overlay nodes to support routing and message switching, which are basic services required in lower levels of an overlay framework. Projects such as CANS [24] and Ninja [23] support the deployment of services along an overlay path to adapt data streams. But these systems are based on ad hoc implementations and do not provide *generic* and *extensible* frameworks for composing new adaptive distributed services. Dynamic service composition projects, such as SpiderNet [19], GridKit [77], and DSMI [78], do address adaptive composition of distributed services. However, each of these projects focuses on different aspects of the problem. For example, SpiderNet provides algorithms to find, select, and adapt distributed service paths; but it does not provide a software infrastructure. GridKit applies an adaptive middleware framework to support overlay services; but it does not provide an architecture to design autonomic communication services. DSMI provides management of resource-aware stream processing, but not a framework to compose and deploy autonomic communication services.

Hence, there is need for a solution that addresses all major aspects of the problem within an integrated design, and from the application layer down to the distributed infrastructure, transparently to the clients. We argue that such an integrated infrastructure can greatly facilitate the development and deployment of future autonomic communication services. The proposed Service Clouds model and architecture represent a new approach to the aforementioned concerns, combining cross-layer and cross-platform adaptation techniques to support autonomic communication.

Chapter 3

Service Clouds Infrastructure

The Service Clouds infrastructure is intended to support the design and deployment of autonomic communication services. In this chapter, we first discuss requirements and challenges for realizing such an infrastructure and state the goals in our design. We introduce the Service Clouds system model and present the architecture. Finally, we describe the implementation of a Service Clouds prototype.

3.1 Requirements and Challenges

As the cyberinfrastructure becomes increasingly complex, the need for autonomic services [86] is also increasing. An essential part of autonomic systems is dynamic creation and reconfiguration of services, transparently to end applications, in response to user requests and changes in the computing environment. With the success of overlay routing and peer-to-peer systems [18, 69, 71], researchers are studying overlay-based frameworks to support communication services [19, 23, 24, 78, 79]. Since the network environment and needs

of the clients vary continuously, communication services need to automatically adapt to changes. Composing distributed communication services requires integration of several complex tasks, including algorithms to compose service paths ¹, protocols to route and process data streams, and dynamic reconfiguration of services.

To date, most research in this area has focused on individual aspects of autonomic communication services, but has not addressed the need for a model and architecture to support the integration of services within a single framework. Below, we describe the research challenges in designing such an infrastructure, and identify those issues we have focused on in our studies.

- **Extensible Architecture.** An approach that supports design and deployment of various services integrated within a framework needs to identify required building blocks and describe how to organize constituent components. The framework has to be generic and extensible in order to be useful for several different services and application domains, and enable developers to integrate services together.
- **Cross-Layer, Cross-Platform Cooperation.** A distributed autonomic infrastructure involves collaboration of multiple software components across layers of a single platform and across participating nodes. An autonomic communication framework needs to facilitate both cross-layer and cross-platform collaborations in order to support autonomic services.
- **Data Stream Adaptation.** High-quality delivery of data streams while processing

¹ Here, we use the term *service path* loosely; collection of nodes deploying a service path form a graph that may not be a path according to the theoretical definition.

them to satisfy client needs and to adapt to a changing environment is a major goal of communication services. An overlay-based communication framework must facilitate adaptation of both data routing and content, in both wired and wireless networks.

- **Overlay Algorithms.** Supporting any overlay-based system are algorithms that locate data processing services, probe the network, and find overlay paths for data streams. An overlay-based framework must support plugging in different algorithms and integrating them with other services, as well as providing mechanisms to support development of various probing algorithms.
- **Service Composition.** An autonomic infrastructure composes services based on user requests, and adapts them as user needs and environmental conditions change. Service composition involves locating suitable places for elements of a distributed service, instantiating service elements (if they are not already running), and reconfiguring service elements. The infrastructure must provide generic algorithms and protocols to compose and adapt services.
- **Self-Management.** An integral part of any autonomic system is self-management. An autonomic communication infrastructure must manage and execute services with minimal human intervention. An integrated framework needs to have several self-* properties, such as self-configuration, self-healing, self-optimization, and self-adaptation.
- **Transparency.** To use autonomic services, a client process needs to interact with the underlying infrastructure. If an application is not implemented directly atop the

infrastructure, it should be possible to transparently shape [87] the application with respect to the existing business code. The mechanisms applied to a particular application depend on the characteristics of its components, including the programming language and the middleware platform used.

While we address all of the above issues in our work, but we do not focus on transparency; other researchers have addressed this issue and their techniques can be applied to bind clients to the Service Clouds infrastructure [87]. Further, designing an integrated framework goes beyond algorithms and requires software deployment to verify and evaluate acceptable operation of a proposed framework with existing systems and provide design guidelines for developers. Many researchers evaluate their solutions using simulation, emulation, and ad hoc implementations. In many cases, such evaluations can demonstrate the *potential* of a specific approach. However, building prototypes and evaluating them on real testbeds is necessary to understand practical limitations and conditions that may otherwise be overlooked.

In the following sections of this chapter, we introduce a model and architecture to compose autonomic communication services, as well as a prototype toolkit. Our design is based on a number of experimental studies and implementations, presented in later chapters. Building prototype systems has been necessary to design a solid model. Our design is based on an iterative approach: using the model to build systems and using the feedback from deployment exercises to refine the model.

3.2 System Model

The Service Clouds model has its roots in an earlier study [28], where we designed and constructed the *Kernel-Middleware eXchange (KMX)*, a set of interfaces and services to facilitate collaboration between middleware and the operating system. We used the KMX to improve quality of service in video streams transmitted across wireless networks (presented in Section 5.3). Figure 3.1 illustrates an example in which a data stream is routed from a source to a destination, traversing three intermediate hosts. Such a situation can occur in environments, such as mobile ad hoc networks and overlay networks, where hosts are involved in the routing of data. Let us assume in this example that packets are routed via the operating system kernel, but that a problem (e.g., a sudden increase in packet loss rate) has occurred on the link connecting nodes *A* and *B*. A monitoring service detects this problem and triggers node *A* to intercept the data stream in the kernel and redirect it to a middleware layer (vertical cooperation), where it is “hardened” by encoding it with a stronger error control method, such as forward error correction. Node *A* informs node *B* (horizontal cooperation) that it also should intercept the data stream. The modified stream is forwarded to node *B*, which does as directed and passes the stream to middleware for decoding.

Our primary focus in the KMX study was on vertical cooperation and we implemented a rudimentary prototype as a proof of concept. However, the study yielded the concept of a *transient proxy*, whereby a service is instantiated on one or more hosts in the network, as needed, in response to changing conditions. An example is the processing on nodes *A* and *B* in Figure 3.1. Indeed, we view Service Clouds as a generalization of this concept.

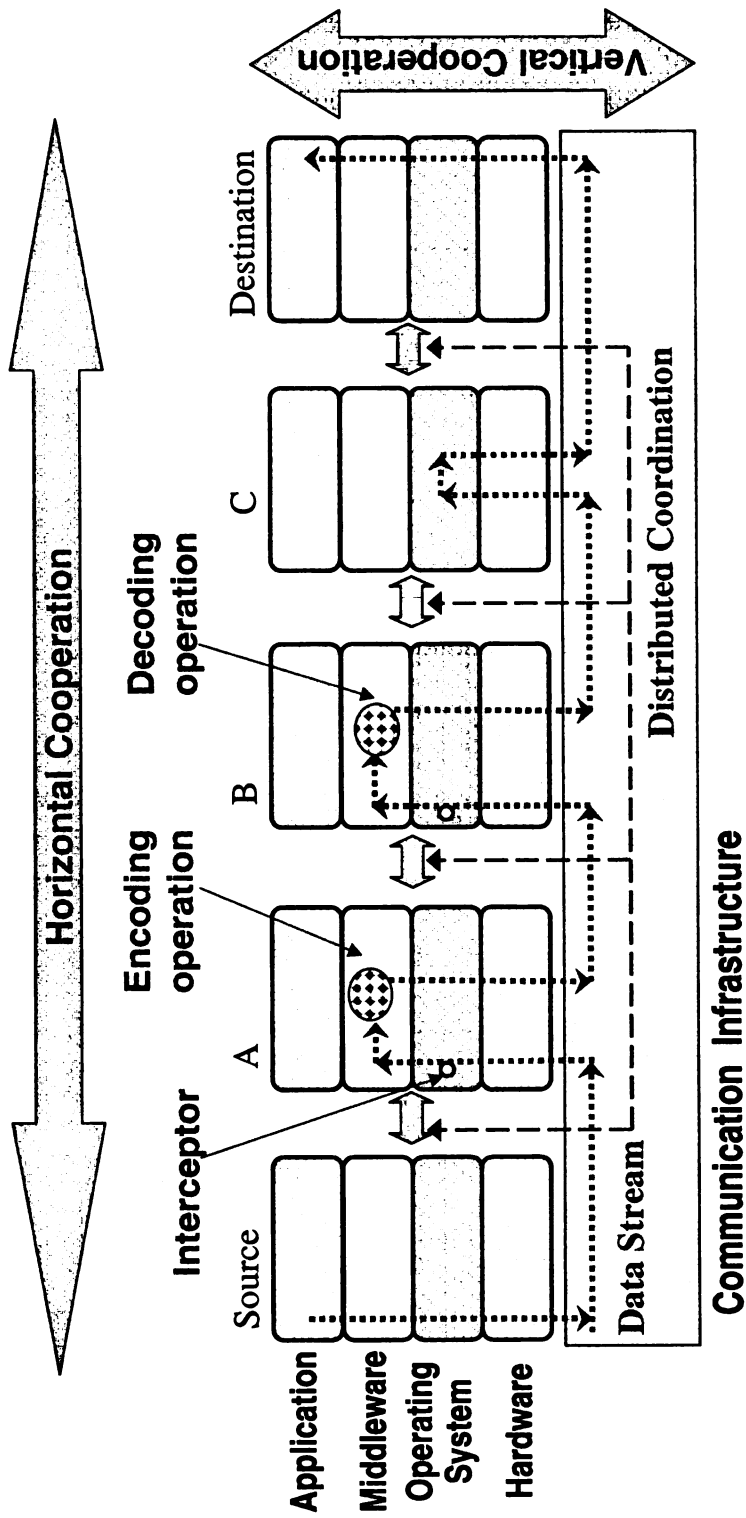


Figure 3.1: Vertical and horizontal cooperation.

The overlay network provides processing and communication resources on which *transient services* can be created as needed to assist distributed applications. As we stated earlier in Chapter 1, in Service Clouds, hosts use adaptive middleware, cross-layer, and cross-platform collaboration to realize autonomic behavior. Effectively, this provides a “blank computational canvas” on which services can be instantiated and reconfigured as needed. Further, Service Clouds provides an extensible infrastructure, in which low-level services facilitate construction of higher-level autonomic services.

3.3 Architecture

We designed the Service Clouds architecture with other developers in mind. Figure 3.2 shows a high-level view of the Service Clouds software organization and its relationship to Schmidt’s model of middleware layers [88]. Most of the Service Clouds infrastructure can be considered *host-infrastructure* middleware, as it can be invoked directly by either the application itself or by another middleware layer. The *Application-Middleware eXchange (AMX)* provides interfaces for that purpose and encapsulates high-level logic to drive various overlay services. Distributed composite services are created by plugging in new algorithms and integrating them with lower-level control and data services, as well as with mechanisms for cross-layer collaboration.

Figure 3.3 provides a detailed view of the current Service Clouds architecture. The architecture comprises four main groups of services, sandwiched between the AMX and the KMX layers. The *Distributed Composite Services* layer coordinates interactions among the layers of a single platform and activities across platforms. An essential aspect of this

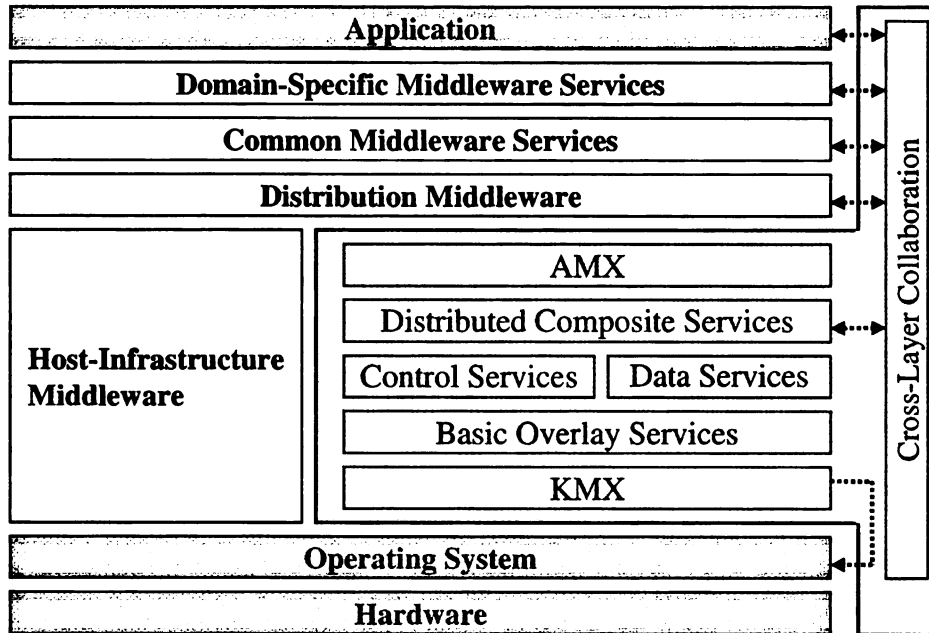


Figure 3.2: Relationship of Service Clouds to other system layers.

architecture is the separation of *data* and *control* connections. Each composite service is decomposed into *Data Services* (e.g., transcoding a video stream for transmission on a low bandwidth link) and *Control Services* (e.g., coordinating the corresponding encoding/decoding actions at the sender and receiver of the stream). *Basic Overlay Services* provide generic facilities for establishing an overlay topology, exchanging status information, and distributing control packets among overlay hosts. Next, we discuss each group of services, starting at the bottom layer and working upward.

Basic Overlay Services include three main types of components: *cross-platform communicators*, *meta-information collectors*, and *overlay routers*. Meta-information collectors at a given node gather system status information, such as current packet loss rate and available bandwidth on each overlay link connected to the node. They also provide services that indicate liveness of service entities running on a node (e.g., by sending heartbeat mes-

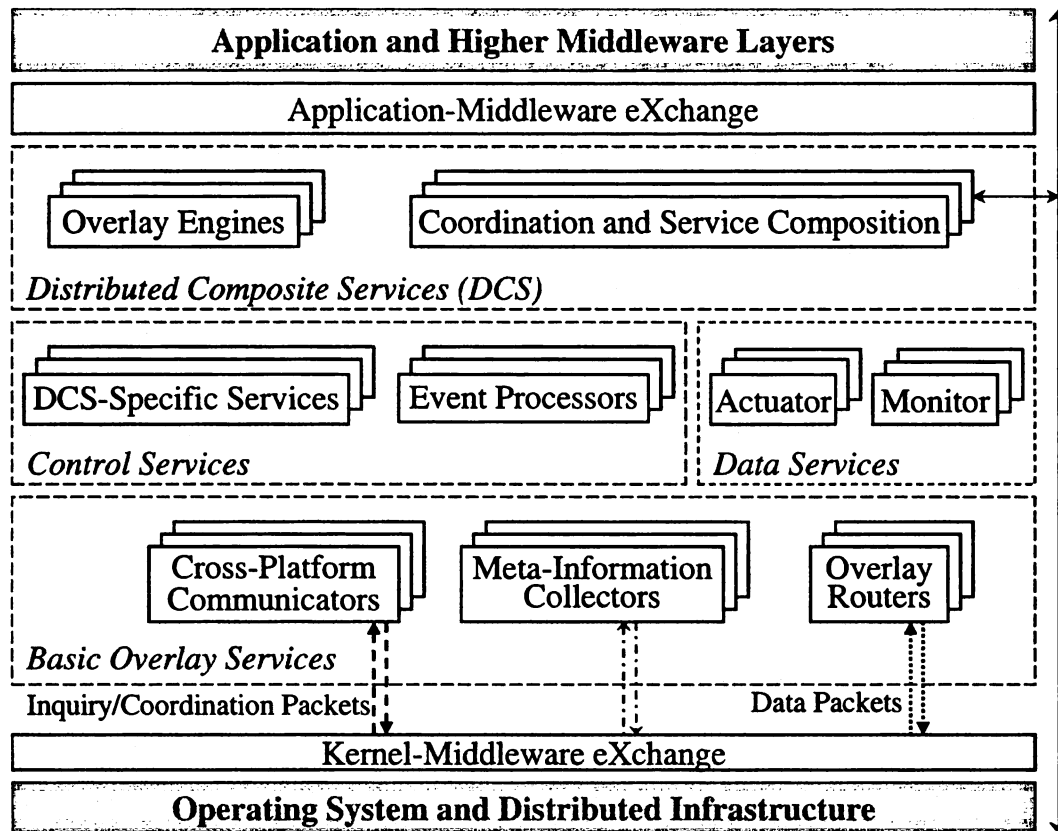


Figure 3.3: The Service Clouds architecture.

sages). Cross-platform communicators send and receive inquiry packets that carry meta-information across platforms in a distributed system. They provide information such as status of resources and availability of services on a node. An overlay router component forwards data packets among overlay nodes and supports their interception for intermediate processing. As well as implementing typical routing protocols, such as source routing, routers can instantiate transient proxies that relay connections in a particular service, for example, managing UDP and TCP connections in a video conferencing session.

Control Services include both *event processors* and *DCS-specific services*. An event processor handles specific control events and inquiry packets, for example, extracting information useful to multiple higher-level services. An event processor can also perform

intermediate processing, for example, constructing statistical information on network conditions. On the other hand, a DCS-specific service implements functionality tied to a particular high-level service, for example, an application-specific probing service.

Data Services are used to process data streams as they traverse a node. *Monitors* carry out measurements on data streams. The metrics can be generic in nature (e.g., packet delay and loss rate) or domain-specific (e.g., jitter in a video stream). *Actuators* are used to modify data streams, based on the information gathered by monitors or by explicit requests from higher-level services; they may also have limited decision making capabilities. Consider the example from Figure 3.1. An actuator can apply FEC filtering on a data stream according to a predefined set of rules and current conditions. We differentiate two types of actuators: *local adaptors* and *transient proxies*. A local adaptor operates on a data stream at the source and/or the destination. For example, a local adaptor operating on a mobile node may handle intermittent disconnections so that applications running on the node do not crash. Transient proxies are adaptors that manipulate data streams at intermediate nodes. For example, a transient proxy at the wireless edge of the Internet can intercept a stream carrying sensitive data and re-encrypt it before it traverses a (potentially less secure) wireless channel. The enabling mechanism for data adaptation is interception, which is usually transparent to the application. Interception at the end nodes can be achieved in several ways: modifying system libraries, using adaptive middleware, or transparently *shaping* existing applications to exhibit new behavior [89].

The Distributed Composite Services unit includes two types of components: *overlay engines* and *coordination and service composition*. An overlay engine executes a particular distributed algorithm across overlay nodes. Examples include building and maintaining a

multicast tree for content distribution, establishing redundant overlay connections between a source and destination, and identifying and configuring relay nodes to improve throughput in bulk data transfers. Coordination and service composition refers to overseeing several supporting activities needed to compose distributed services and to realize vertical and horizontal cooperation. For example, a coordinator can accept a service request and determine to use a particular overlay algorithm, set the update intervals in meta-information collectors for gathering information (which affects the overhead and accuracy of the gathered data), and configure a kernel-level module to pre-process data packets and discard useless ones early to avoid unnecessary load and resource consumption.

Finally, as we mentioned earlier, the Service Clouds architecture includes two bounding strata: the KMX and the AMX layers. The KMX layer provides a platform-independent interface to OS- and network-specific services and information. For example, the KMX facilitates to provide information on local system resource availability, report the status of connections to other nodes, and intercept data streams at the OS level and redirect them to transient proxies (if the operating system provides such services). The AMX layer provides an interface that enables applications and higher-level middleware layers to interact with the Service Clouds infrastructure. This interaction includes accepting and replying service requests, and receiving and delivering data.

3.4 Prototype Implementation

Building and testing a prototype of Service Clouds have been necessary to identify several key practical issues that must be addressed in distributed service-oriented frameworks. In

this section, we describe the prototype framework and components based on the implemented architecture.

Figure 3.4 depicts the Service Clouds prototype. This prototype is written primarily in Java, but can incorporate modules written in other languages by using the Java Native Interface (JNI). Below, we provide a high-level overview of several components implemented within the prototype framework and the interaction among them. The case studies presented in the next chapters provide a more detailed description of these components and particular instances of them.

The prototype software has a main driver, which deploys coordination and service composition. It provides *primitives* such as those for reading configuration files containing the IP addresses of overlay nodes and the overlay topology, instantiating a basic set of components as threads, and configuring the components according to the default or specified parameter values. Example parameters include the interval between probes for monitoring packet loss, the maximum number of hops an inquiry packet can travel, and an assortment of debugging options. The *Service Gateway* component implements simple protocols to accept and reply to service requests. The *Service Composer* component implements mechanisms for composing a service path. For example, it can use a *Relay Manager* to instantiate and configure a UDP relay.

Relays and *Routers* deploy data routing mechanisms and enable the infrastructure to intercept and process the stream in the Data Services unit. Routers implement typical routing protocols such as source routing. Relays are transient proxies that are dynamically instantiated for a specific service session and last only for the duration of the session. For example, in a video conferencing service, a number of relays handle UDP (for audio and

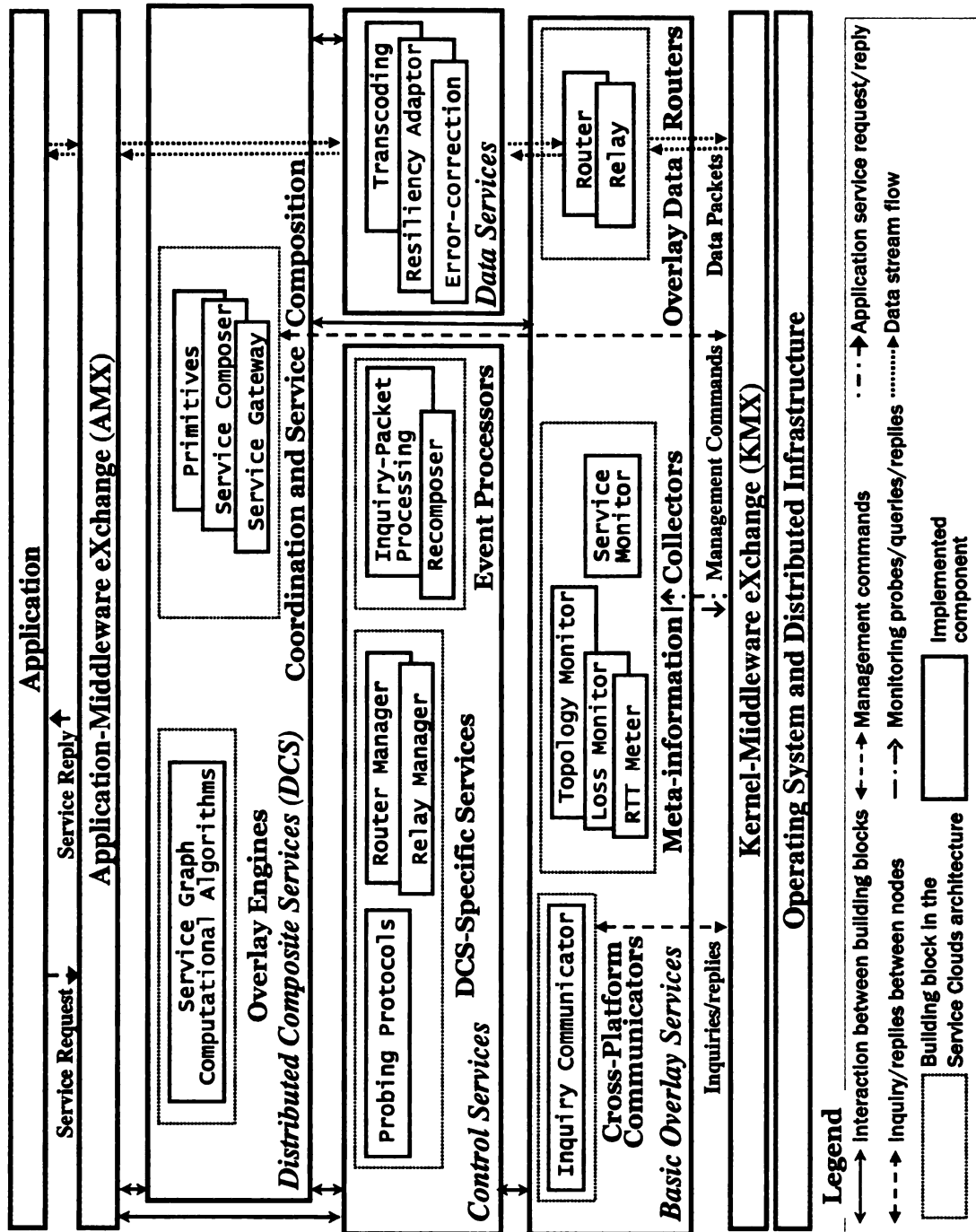


Figure 3.4: The Service Clouds prototype.

video packets) and TCP (for application control lines) connections used by the end node applications. Such intermediate transient services are created by sending an appropriate request to the Service Composer unit on a suitable remote node, which spawns the necessary threads. Example data processing components include those that apply error-correction filters, send and receive data on backup paths for resiliency, and transcode multimedia streams to overcome limitations on mobile clients. Dynamic adaptation techniques, such as MetaSockets [90], can be used to insert and remove data filters at run time.

Components such as meta-information collectors implement the *singleton pattern* [91], which means that different overlay services refer to and use the same instantiation of a monitoring service. Implemented examples include topology monitor, link loss monitors, and inter-node round-trip time (RTT) monitor. This design facilitates sharing of component services among more complex services. *Service Monitors* use mechanisms, such as sending heartbeat messages, to announce liveness of a service entity or node. The *Recomposer* component tracks activity of distributed service components using Service Monitors. Upon detecting a failure, it initiates a self-healing operation that recomposes the service path and restores communication.

We have implemented a number of overlay algorithms and supporting probing mechanisms that together compute a service graph. Those will be presented in case studies described in the following chapters. The inquiry packets carry the probes, which are handled in the inquiry communicator and processing building blocks. The format of inquiry packets exchanged among nodes is XML. Figure 3.5 shows a snippet of a sample inquiry packet used in one of the case studies (MCON in Section 4.3). In that case study, the probing protocol gathers physical topology and link loss status information throughout the overlay

network, which are analyzed by an algorithm to find a set of high-quality overlay routes. We use DOM [92] to access and manipulate the XML content of the inquiry packets. Although DOM may not be as fast as the other alternatives, such as SAX [93], it provides a convenient interface for inspecting and modifying XML fields, and benchmarking our prototype indicates its performance is satisfactory. Further, the emergence of efficient XML technologies, such as binary XML format [94], also addresses concerns about bandwidth consumption and the speed of processing XML documents.

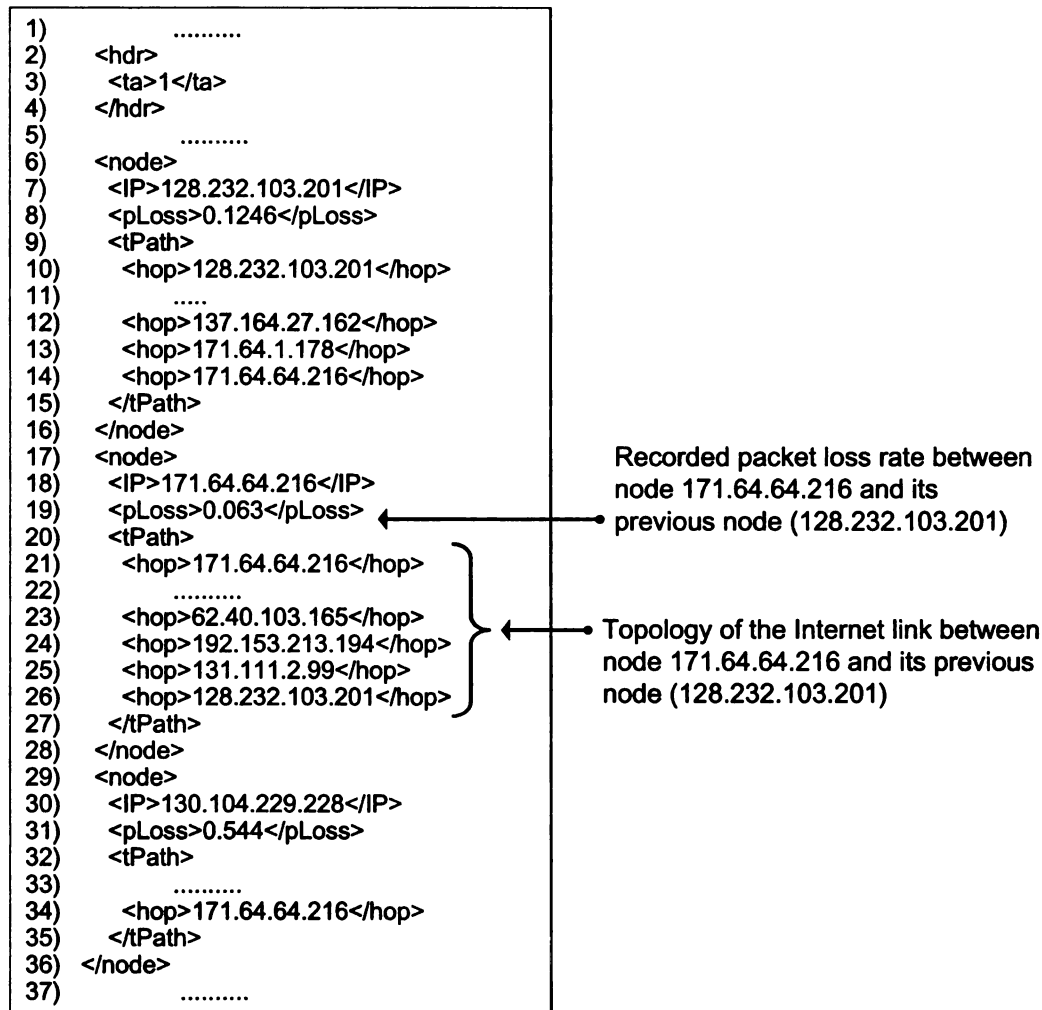


Figure 3.5: Snippet of a sample XML inquiry packet.

To run the prototype and automated tests, the prototype includes a collection of scripts

to support execution on PlanetLab [95], a distributed Internet testbed. These scripts are used to configure nodes, update code on the nodes, launch the infrastructure, setup and run test scenarios, and collect results. To manage the distributed infrastructure, we have used Java Message Service (JMS) [96]. We have implemented a simple set of remote management commands that enable gathering of statistics (e.g., obtaining number of inquiry packets received at each node), to change framework parameters at run time, and to reset or shut down the distributed infrastructure.

Finally, to incorporate a software application to an infrastructure such as Service Clouds, we need to support the interactions between the application and the underlying services. The interactions need to be considered at the design time, during configuration of the system, and at run time. Generally, to support interactive activities in such a service-oriented architecture, we need to: (a) Modify an existing application to use the underlying services if it was not implemented atop the infrastructure. One could modify the application manually, but a better approach is to transparently shape the application with respect to the existing business code. The mechanisms to apply on a particular application depend on its characteristics including the programming language and the middleware platform. (b) Configure and bind the application and services. The infrastructure provides the application with ways to customize specific configurations and to request and maintain services. Only after the service binding is established, can an application start using the underlying services. In the Service Clouds prototype, we have used transparent shaping [87] and autonomic service specification [97] techniques, developed in our lab, to enable existing applications on mobile clients to execute atop the infrastructure.

Effectively, the Service Clouds prototype framework provides the “glue code” and a

repository of components as a toolkit to support rapid deployment of services, or to be used as a guide to developing new services. The entire source code and additional documentation are available at the RAPIDware project homepage [98]. We emphasize that the purpose of this prototype is merely to identify different aspects of the problem and conduct a requirements analysis. Therefore, we have implemented a limited set of features. Building the prototype has been helpful in revealing salient issues for designing a more complete Service Clouds framework.

Chapter 4

Deep Service Clouds: Experimental Evaluation on the Internet

We have used the Service Clouds model to investigate deployment of enhanced communication services in *Deep Service Clouds*, which executes within an Internet overlay network, such as wired nodes on the PlanetLab [95] testbed. We used the Service Clouds architecture to deploy two different services: TCP-Relay and MCON. TCP-Relay involves automatic and transparent deployment of TCP relays, whereby data streams are redirected through a selected overlay node. Experimental results show that for bulk data transfers, use of a TCP relay can reduce transfer time substantially, relative to the native TCP/IP connection. MCON is a service for creating resilient network connections by dynamically establishing one or more high-quality backup overlay paths between a source and destination. Experimental results show effectiveness of using physical network topology in finding high-quality backup paths. Conducting these case studies helped redefine the Service Clouds architecture, and demonstrate the usefulness of the Service Clouds model.

This chapter is organized as follows. Section 4.1 introduces the experimental testbed and the prototype instantiation for the presented case studies. Sections 4.2 and 4.3 describe the operation and implementation details of the TCP-Relay and MCON services, respectively, and present and discuss empirical results. Section 4.4 discusses related work, and Section 4.5 concludes this chapter.

4.1 Experimental Testbed and Implementation

Figure 4.1 shows a geographical map of nodes used in the case studies on PlanetLab [95], an Internet research testbed comprising hundreds of Linux-based Internet nodes distributed all over the world. Below, we briefly describe the prototype implementation for Deep Service Clouds. We describe and discuss details of each case study service in its corresponding section.



Figure 4.1: Map of PlanetLab nodes used in the Deep Service Clouds experiments.

Figure 4.2 depicts the instantiation of the prototype for Deep Service Clouds, including specific components used to construct the TCP-Relay and the MCON services. Components labeled with an *R* are specific to the TCP-Relay implementation, while those labeled with an *M* are specific to the MCON implementation. The rest of the implemented components are not service-specific. The *AMX* interface to the infrastructure has been implemented using local TCP sockets in this instantiation of the prototype. The overlay engines implement the computational algorithms for TCP-Relay and MCON. The components used in deploying the TCP-Relay and the MCON services allow these services to monitor the overlay network, perform distributed probing, and establish high-quality service paths.

In TCP-Relay, probing is simple and includes only collection of round-trip time (RTT) measurements. The *RTT Monitor component* directly measures RTT between a local node and all neighbors by periodically sending “ping” messages in a low-priority background thread. Another background thread implements a simple UDP-based protocol for requesting the RTT between any two remote nodes, to maintain a local view of delay conditions across the entire overlay. To create a relay at run time, the *Relay Establish* component communicates with the *Relay Manager* on a remote node, which dynamically instantiates and configures a *TCP Relay*.

In MCON, the *Path-Establish* and *Path-Explore* components implement the probing protocol to find a backup overlay path once a primary path has been established. The *Inquiry Communicator* and *Inquiry-Packet Processing* components process the XML probe messages and extract information for the probing protocol. The *Resiliency Adaptor* component supports duplicating UDP packets on a backup path on the source node and receiving a single copy at the destination node. The *Topology Monitor* and *Loss Monitor* components

provide topology and loss rate information between a node and its neighbors. To gather topology information, this implementation invokes the Linux/Unix “tracepath” tool. The packet loss rate to the neighbors is measured by sending and receiving beacons and acknowledgements. The *Source Router* component implements a source routing mechanism to route data packets through the overlay network along a predetermined path.

4.2 Case Study: TCP-Relay

Early overlay networks provided UDP-based routing and data forwarding services, essentially using application-layer entities to emulate network-layer functionality. However, a set of recent studies indicate that application-level relays in an overlay network can actually improve TCP throughput for long-distance bulk transfers [99–101]. Specifically, due to the dependence of TCP throughput on round-trip time (RTT), splitting a connection into two (or more) shorter segments can increase throughput, depending on the location of the relay nodes and the overhead of intercepting and relaying the data. To develop a practical TCP-Relay service, key issues to be addressed include identification of promising relay nodes for individual data transfers, and the dynamic instantiation of the relay software. In this case study, we use the Service Clouds infrastructure to construct such a service, and we evaluate the resulting performance.

4.2.1 Basic Operation

Figure 4.3 illustrates the use of a TCP relay for a data transfer from a source s to a destination d . Let R_{sd} and p_{sd} denote the round-trip time (RTT) and loss rate, respectively, along

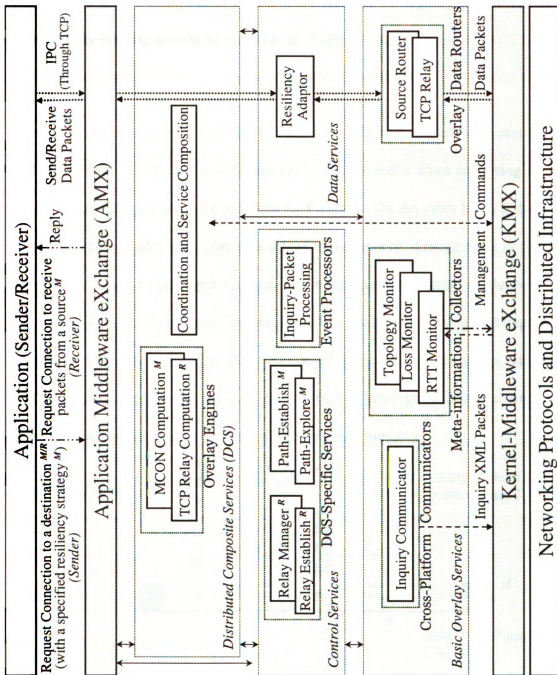


Figure 4.2: The Deep Service Clouds prototype.

the default Internet path connecting s and d . The performance of a large TCP transfer from s to d is mainly determined by TCP's long-run average transmission rate, denoted T_{sd} , which can be derived using one of several well-known formulas [102, 103]. The simplest of these is the formula derived by Mathis et al. [102]:

$$T_{sd} \approx \frac{1.5 M}{R_{sd} \sqrt{p_{sd}}},$$

where M is the maximum size of a TCP segment. Since T_{sd} is inversely proportional to RTT, TCP flows with long propagation delays tend to suffer when competing for bottleneck bandwidth against flows with low RTT values. On the other hand, if the same transfer is implemented with two separate TCP connections through node r , then the approximate average throughput T_{srd} will be the minimum throughput on the two hops, $T_{srd} = \min(T_{sr}, T_{rd})$. In practice, the rates of the two sessions may be coupled imperfectly through back pressure induced by TCP flow control. Relay r is, in principle, a suitable relay for the transfer in question if $T_{srd} > T_{sd}$. In practice, however, the potential throughput improvement generally has to be above some minimum threshold.

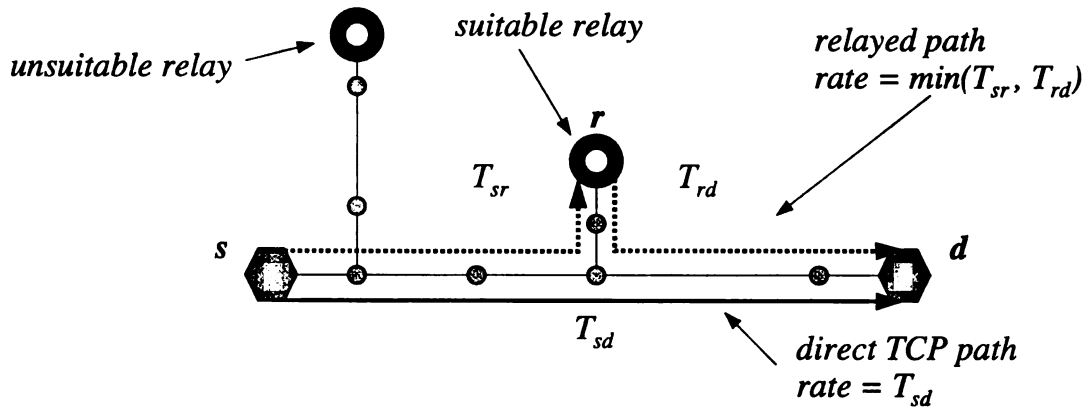


Figure 4.3: An example of a TCP relay.

If more than one suitable relay can be chosen, then a routing decision is required to determine which of the possible relays will yield the best performance. Typically, a well-

chosen relay will be roughly equidistant between s and d and has to satisfy:

$$R_{sr} + R_{rd} < \gamma R_{sd},$$

where $\gamma > 1$ is a small stretch factor. In this case study, we limit consideration to a single relay. However, routing a connection through a sequence of relays can further improve performance, with marginally diminishing improvement as the number of relays increases.

A TCP-Relay service must satisfy several requirements. First, the path properties such as RTT and loss rate used to predict TCP throughput must be measured continuously and unobtrusively. Since no overlay node can directly measure all paths, measurement results must be distributed across the overlay as needed. Second, the initiator of the transfer must use the measurement results to predict the TCP throughput for all possible choices of relay and select the best option (which may be the direct Internet path if no good relay exists). Third, the initiator must signal the chosen relay node to set up and tear down a linked pair of TCP connections. Finally, the relay must forward data efficiently, with as little overhead as possible. As we will see below, these requirements map neatly to the separation of concerns among particular component types in the Service Clouds architecture. Furthermore, the modularization provided by the Service Clouds architecture provides clear implementation points for future performance improvements and functional enhancements.

4.2.2 Implementation Details

Let us refer back to Figure 4.2. Several components collaborate to address the four requirements described above. First, the *RTT Monitor* component encapsulates the collection and

distribution of delay measurements. Other local components query RTT Monitor to obtain RTT information. Second, the *TCP Relay Computation* component encapsulates the computation required to predict TCP throughput and find optimal relays. It exposes an interface for relay selection and relies on Meta-Information Collectors such as the RTT Monitor to provide input for its computation. Third, the *Relay Establish* and *Relay Manager* components collectively implement the signalling protocol to set up and control a relay. Relay Establish exposes an interface by which an end node application can initiate the setup process (via AMX). The Relay Manager operates at the relay node to allocate resources in response to requests for relay service. Finally, the *TCP Relay* component implements data forwarding at the relay node, running in a transient thread instantiated by the Relay Manager for the duration of a particular transfer. This component currently performs data forwarding in user space by transferring data between two sockets. However, a more advanced implementation might make use of KMX to invoke kernel-level support for more efficient data forwarding.

Figure 4.4 illustrates an example of the distributed operation of the TCP-Relay prototype, in which an application sender (on the left) uses the service to transfer data via a relay (on the right) to a receiver (not shown). The interactions among the four Service Clouds components are numbered to indicate the sequence of steps required to complete the transfer; below, we describe this sequence of steps.

The sender issues a request to AMX for a connection to a particular receiver address (step 1). Upon receiving the application request, the AMX layer invokes the TCP Relay Computation component to find a suitable relay (step 2). The TCP Relay Computation engine queries the RTT Monitor component to obtain the RTTs between the node and all of

its neighbors as well as the RTTs between its neighbors and the intended receiver (step 3), using this information to select a relay node. The address of the chosen relay is returned to AMX (step 4).

Having been informed of the relay, AMX invokes the Relay Establish component to request service (step 5), causing Relay Establish to signal the Relay Manager at the relay node (step 6). The Relay Manager spawns a thread in which to run a new instance of the TCP Relay component (step 7). Using socket system calls, the TCP Relay opens a TCP connection to the receiver and also begins listening for an incoming connection from the sender (step 8). The Relay Manager remains blocked until the connection to the receiver is established; once unblocked, it signals Relay Establish at the sender, indicating the port on which the newly created TCP Relay can be contacted (step 9). AMX receives control along with the relay port number (step 10) and opens a TCP connection to the relay (step 11). AMX also creates an inter-process communication (IPC) port on which it will accept data and returns control to the application (step 12).

With the infrastructure now in place to support the transfer, the application connects to the IPC port and begins to send data (step 13). AMX forwards the data over the TCP connection created in step 11. The TCP Relay component reads this data and forwards it over the connection created in step 8. When the data transfer is complete, the application closes the IPC socket, triggering AMX to close its connection to the relay, which in turn closes its connection to the receiver. When all connections are closed, TCP Relay terminates its thread, thereby removing any transient state associated with the transfer.

To show how the TCP-Relay service operates within the prototype framework, we briefly describe selected portions of the program. Figure 4.5 excerpts code from the TCP-

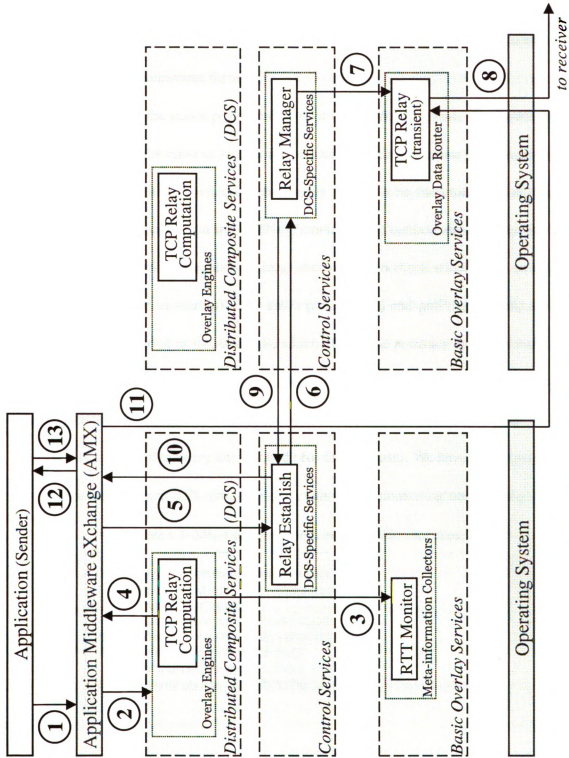


Figure 4.4: An example of TCP-Relay run-time operation.

Relay implementation. The `Constructor` class (lines 9-19) obtains instances of objects for RTT monitoring, relay management, interaction with the application, and connection to the JMS server. The `tracker` methods (lines 22-30) start necessary service elements, each comprising several concurrent threads.

Table 4.1 shows the major public methods of the `RttMonitor` class. The `getInstance()` method instantiates the class or returns the instance if it already exists (the singleton pattern). The `init(Vector)` method specifies overlay nodes to be monitored. The `tracker()` method starts monitoring, and the `isAllNeighborsPinged()` method shows if the monitoring service has pinged all neighbors at least once (useful to check whether the service has completed booting at its startup). The `setPingInterval(int)` and `getPingInterval()` methods set and get ping interval in milliseconds, which affects the accuracy and overhead of the monitoring service. Finally, the `getRTT(String)` and `getRTT(String, String)` methods return measured RTT from the current node to a neighbor, or between two remote nodes (the class implements a protocol to query another node for this purpose). We have also implemented procedures that measure RTT upon a request, rather than monitoring nodes continuously.

Table 4.1: Major public methods of the `RttMonitor` class.

```

1 public static RttMonitor getInstance();
2 public final void init(Vector vNeighborhood);
3 public final void tracker();
4 public boolean isAllNeighborsPinged();
5 public static int setPingInterval(int msecs);
6 public static int getPingInterval();
7 public final double getRTT(String neighborIP);
8 public final double getRTT(String neighborIP, String destinationIP);

```

To create a TCP relay on a remote node, the `RelayEstablish` class provides this method:

```

public static int setupRelay(String destiNode, int destiPort, String relayNode);

```

```

1 public class uaftcp {
2     private RttMonitor rttmonitorObj;
3     private TCPRelayManager relaymanagerObj;
4     private TCPRelayAppDriver appdriverObj;
5     private static SimpleAsynchConsumer jmsCommObj;
6     private static Vector vNeighbors = new Vector();
7     ...
8     // Constructor
9     public uaftcp(____) {
10         ...
11         if (initNodes(____) <= 0)
12             {...}
13         rttmonitorObj = RttMonitor.getInstance();
14         relaymanagerObj = TCPRelayManager.getInstance();
15         appdriverObj = new TCPRelayAppDriver();
16         if (UConfigVars.isJmsFeature())
17             {jmsCommObj = new SimpleAsynchConsumer();}
18         ...
19     } // end uaftcp() constructor
20     ...
21     // Starts all the required threads.
22     public void tracker() {
23         try {
24             if (UConfigVars.isJmsFeature()) // JMS
25                 {jmsCommObj.tracker("jms/Topic");}
26             rttmonitorObj.tracker(); // Latency monitor
27             relaymanagerObj.tracker(); // Relay manager
28             appdriverObj.tracker(); // Activating AMX
29         } catch (Exception e) {...}
30     } // end tracker()
31
32     // Main method
33     public static void main(String[] args) {
34         try {
35             ...
36             uaftcp uafmconObj = new uaftcp(____);
37             uafmconObj.tracker();
38         } catch (Exception e) {...}
39     } // end main()
40 } // end uaftcp

```

Figure 4.5: Snippet of the TCP-Relay service class.

which sends a request to the overlay node (`relayNode`) and specifies the destination address and port number. On the relay node, the service composer uses the following method from the `TCPRelayManager` class to instantiate a TCP connection to the destination:

```
public static void setupTCPRelay(final String destinationIP, final int destinationPort);
```

which also instantiates a thread that accepts a TCP connection from the service requester (on a pre-defined port in the current implementation) to pass its data towards the destination on the instantiated connection. Other implemented classes provide easy-to-use interfaces in a similar manner.

4.2.3 Empirical Results and Analysis

To evaluate the TCP-Relay service, we constructed an overlay network comprising 10 PlanetLab nodes, listed in Table 4.2. In these tests, all transfers originate and terminate at overlay nodes, possibly using other overlay nodes as relays. As we expect the TCP-Relay service to be particularly useful for long-distance transfers, we selected a set of geographically dispersed nodes across four continents, as illustrated in Figure 4.1 earlier in this chapter. We conducted two experiments on this network—a global measurement of the TCP-Relay performance for all sender-receiver pairs in the overlay, and a more thorough evaluation of the service for one particularly well-performing case. We now describe both experiments in detail.

To understand the network-wide behavior of the TCP-Relay service, we conducted an exhaustive set of data transfers between all possible sender-receiver pairs. Since any PlanetLab node can reach any other over the Internet, the overlay network for our experiment can

Table 4.2: List of nodes in the TCP-Relay experiment.

ID	Node Name	Location
0	planet-lab-1.csse.monash.edu.au	Australia
1	plab1.cs.ust.hk	Hong Kong (China)
2	planetlab1.cs.cornell.edu	USA - NY
3	planet1.cs.ucsb.edu	USA - CA
4	freedom.ri.uni-tuebingen.de	Germany
5	planet1.pittsburgh.intel-research.net	USA - PA
6	planetlab01.cs.washington.edu	USA – WA
7	planetlab3.uvic.ca	Canada
8	planetslug1.cse.ucsc.edu	USA - CA
9	planetlab2.polito.it	Italy

be represented as a complete directed graph with ten vertices and edges representing the 90 possible sender-receiver pairs. For each sender-receiver pair, any one of the eight remaining overlay nodes is a candidate relay. To evaluate the performance of the prototype with respect to a particular pair, we performed two back-to-back data transfers of equal size—one via a direct TCP connection and another using the TCP-Relay infrastructure—recording the throughput achieved by the relayed transfer divided by that of the direct transfer. We refer to this quantity as the *improvement ratio*.

For transfer sizes ranging from 2 to 64MB, we repeatedly cycle through all possible sender-receiver pairs collecting one measurement of improvement ratio per pair per cycle. We ran this experiment continuously for a period of roughly two weeks. In some cases, measurements could not be taken due to the temporary unavailability of a PlanetLab node or an incomplete data transfer. Note that there is no guarantee that the same relay will be selected across all measurements for the same sender-receiver pair. While we observed

some changes in the selected relay, we also observed several sender-receiver pairs with highly stable relays that provided a significant improvement in performance.

Analysis of the collected data from the experiment described above shows that for 49 out of 90 possible sender-receiver pairs the TCP-Relay service infrastructure selected a relay over the direct path in at least one experimental run. It is not surprising to find many cases in which no relay was selected, as our overlay includes several nodes on both coasts of the United States all connected by the Abilene Internet 2 backbone. For pairs selected from this subset, it is difficult to find a relay that can improve performance measurably over a direct connection. In such cases, the improvement ratio measurement reflects the overhead of an unsuccessful search for a relay; this overhead was negligible for transfer sizes we considered.

Figure 4.6 shows experimental results for a transfer size of 8MB, plotting the mean, maximum, and minimum measured improvement ratio for the 32 sender-receiver pairs in which a relay was chosen in at least 4 sample runs. The pairs are rank-ordered along the x-axis in descending order of the mean improvement ratio.

In half of these cases (16 pairs) the use of a relay improved the average throughput—equivalently, the average transfer time—by at least 15%. On the other hand, in roughly 20% of the cases where a relay is used, the average throughput is degraded. We see from the maximum and minimum improvement ratios that there is significant variability in the relay performance such that even pairs for which the average performance was degraded often found at least one good relay. We observed that the relay selection for the best performing cases was highly stable and that improvement was consistently significant. We note, however, that the maximum improvement ratios on the order of 3.0 cannot be attributed

to the relay effect alone; these values reflect measurements in which the direct connection achieved unusually poor throughput, possibly due to high packet loss rate.

The mixed results presented in Figure 4.6 require some interpretation. In terms of the fraction of cases in which the mean improvement ratio was greater than 1, our observations are roughly consistent with those of a similar experiment conducted by Swamy [99]. However, we believe that the performance of the TCP-Relay service can be improved considerably. As the preliminary goal of our prototype was to explore the Service Clouds architecture, rather than a complete analysis of the TCP-Relay approach, we implemented a fairly basic service, omitting several features that would allow much better prediction of TCP throughput. The extensibility of the Service Clouds architecture supports incorporating existing services and tools to improve the TCP-Relay performance. For example, unobtrusive measurement of packet loss and available bandwidth [104] can be readily integrated in the form of additional meta-information collector components. With this richer set of measurements, it will be possible to make more accurate throughput predictions such as those proposed by He et al. [105].

Figure 4.7 shows results for a particular sender-receiver pair in our experiment. The sender (ID 0 in Table 4.2) is in Australia, and the receiver (ID 1 in Table 4.2) is in Hong Kong. For this pair, our prototype service consistently selected a relay on the East Coast of the United States (ID 2 in Table 4.2). Although the location of the relay is counterintuitive, an examination of paths discovered by traceroute offers a straightforward explanation. The default Internet route from the sender to the receiver makes a trans-pacific hop to the United States, then traverses a high-performance backbone network (Abilene) to an inter-ISP exchange point in Chicago (Figure 4.8). A few hops after Chicago is a second trans-pacific

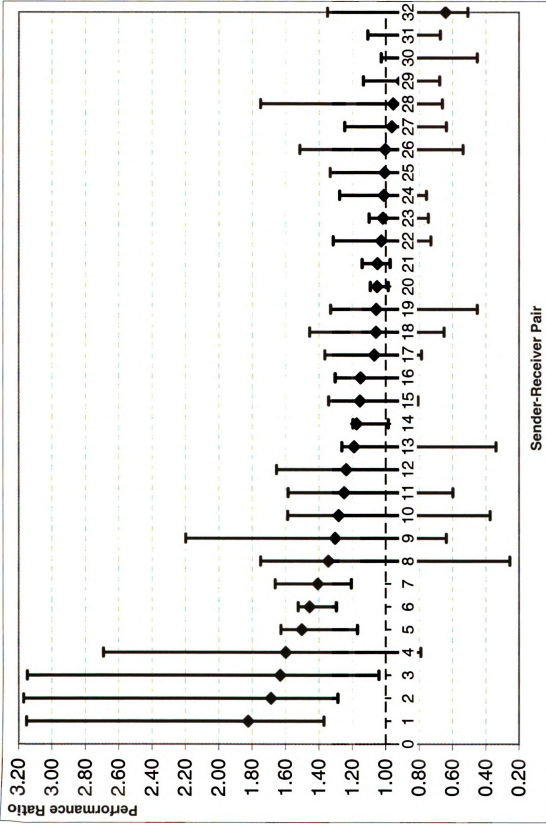
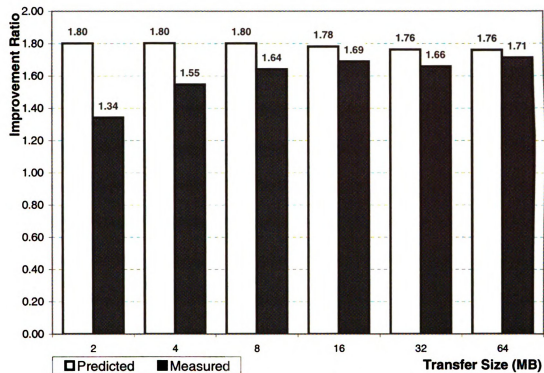


Figure 4.6: TCP-Relay overall empirical results on PlanetLab.

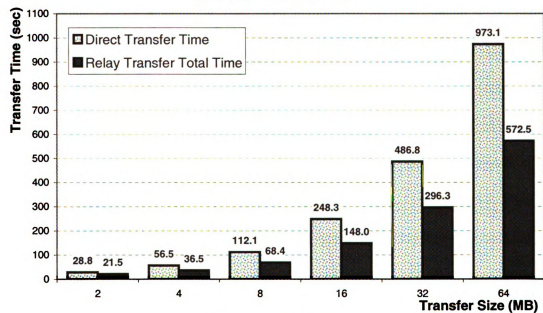
hop to China. Each trans-pacific hop adds roughly 200 msec to the total RTT, placing the Chicago exchange point very close to the midpoint of the route. The chosen relay turns out to be an excellent choice—adding only a 20 msec detour through Abilene before rejoining the direct route in Chicago.

Figure 4.7(a) shows the predicted and observed improvement ratio for relayed transfers of various sizes using the Australia-to-Hong Kong pair. Each observed value is an average of at least 10 measurements. This plot shows that the measured improvements are close to the theoretical prediction. It is worth emphasizing that this performance improvement is entirely attributable to the relay's reduction of TCP feedback delay—without examining the traceroute data, an alternative explanation might have been the discovery of a more direct path not available to the default Internet route due to policy-based routing at the network-layer.

We also observe the amortization of startup latency at larger transfer sizes, which can also be seen in comparisons of the durations of direct and relayed transfers in Figure 4.7(b). The relay transfer total time is the sum of two items: relay setup time and relay data transfer time. *Relay setup time* is the time spent from the moment the sender application sends a request to the overlay service to transfer data to a specified destination, until the overlay service signals the sender to begin the data transfer through the overlay infrastructure (which transfers the data via the best relay). This includes the time taken by the overlay service to gather links status data, select the best relay, and configure the relay dynamically. *Relay data transfer time* shows the duration of transfer through the overlay infrastructure. This duration spans from the time the receiver application gets the first block of data until the data transfer is complete and the connection is closed (we use the same method to measure



(a) average improvement ratio



(b) average transfer time

Figure 4.7: TCP-Relay results for a selected pair.

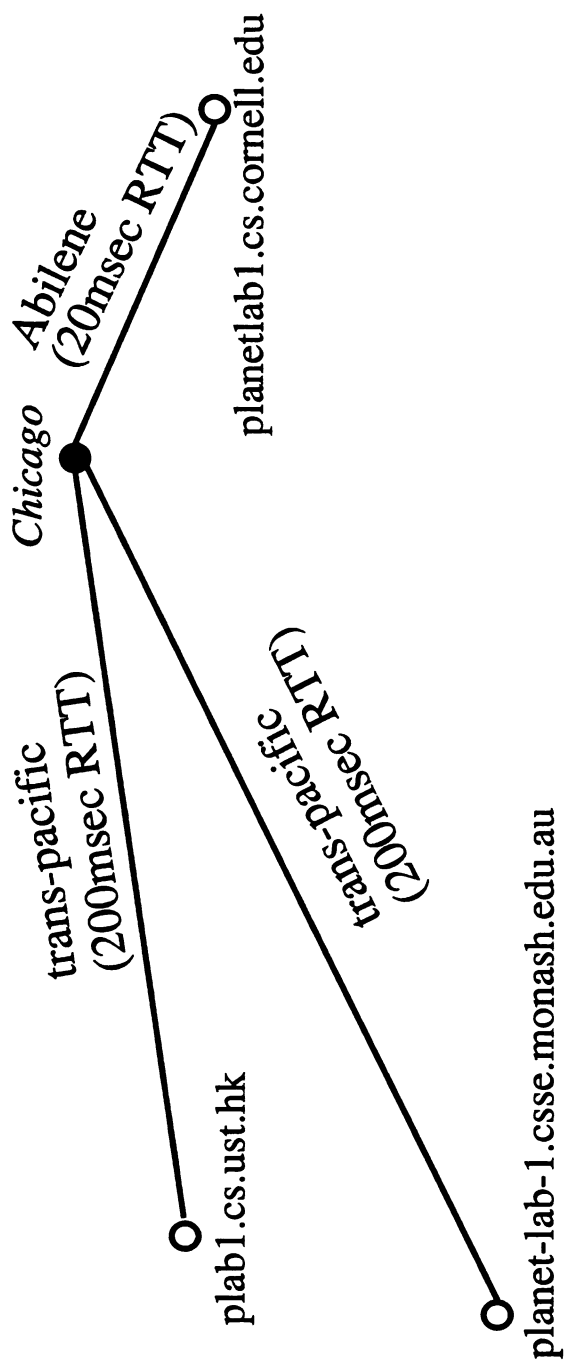


Figure 4.8: Internet path of the TCP-Relay sample.

the direct data transfer time).

The results show that average setup overhead is below 5 seconds, which is insignificant for a bulk data transfer that lasts for several minutes. It also explains why the average performance ratio measured for small transfer sizes is not as close to the expected gain as that of the large transfer sizes. When the direct transfer time is large (above 100 seconds), the 5 second relay setup overhead has little or no effect on the performance gained. Moreover, at 64MB, the largest transfer in our experiment still constitutes a relatively small bulk transfer. Yet even at this scale, we see a significant reduction in transfer time—from approximately fifteen minutes to approximately ten minutes.

Next, we extend TCP-Relay to provide a more robust service by recomposing a service path whenever it fails to satisfy the expected performance or network conditions change significantly.

4.2.4 Adaptive TCP-Relay

Adaptive TCP-Relay provides a robust service by monitoring data transfer rate at run time and adapting the data transfer path when conditions change. If the service detects the composed overlay path may not be satisfying the expected performance, it uses a rule-based decision-making procedure to recompose the overlay path by inserting, removing, or relocating the relay.

Design and Operation. In adaptive TCP-Relay, the service at the sender node (source) monitors data transfer performance in terms of transfer rate and triggers an adaptation procedure if it detects a significant change in the performance. If the data is being transferred

through a relay, the service adapts by relocating the relay function to another overlay node, predicted to be a better relay, or establishing a direct connection if it cannot locate such an overlay node. If the current connection is direct, the recomposition establishes a relay if the services locates a suitable overlay node.

Figure 4.9 shows the basic flowchart of the adaptive TCP-Relay algorithm. Initially, to compose a service path connecting the endpoints, the system looks for an overlay node suitable to act as a relay. If such a node is not found, the data transfer uses a direct TCP connection between the endpoints (*Direct Transfer Mode*). As the direct transfer proceeds, the system continues looking for potential relays that may provide better performance due to changing conditions.

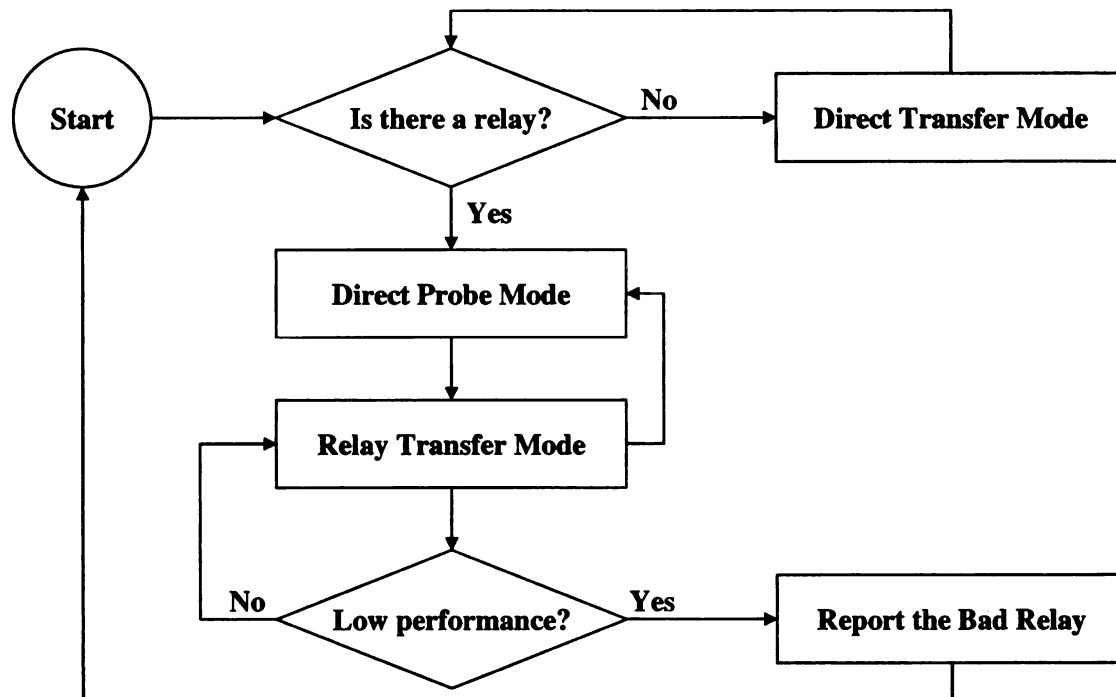


Figure 4.9: Basic flowchart of the adaptive TCP-Relay algorithm.

If the system predicts a high-performance connection via an overlay node, it configures that node as the relay. This prediction may not hold in practice due to factors such as net-

work congestion or low performance of the chosen node (e.g., if it is overloaded). Thus, before transferring data via a selected overlay node as a relay, the system transfers a predetermined amount of data via the direct TCP connection (*Direct Probe Mode*), and measures the performance of the direct connection. Afterwards, the system establishes a connection via the relay to transfer the rest of the data (*Relay Transfer Mode*). Since the conditions on the Internet may change any time, the system keeps monitoring the performance of the composed service path. If the performance of the relayed connection becomes unacceptable, the system marks the relay as a bad one and starts recomposition in order to adapt the service path to the new conditions.

Implementation. To explain the operation of the adaptive TCP-Relay algorithm, we use the flowchart in Figure 4.10. As noted above, the system first transfers a specified amount of data in Direct Transfer Mode and measures the throughput of the direct TCP connection. Next, the system looks for a suitable overlay relay path that is, one with a performance predicted to be higher than the direct connection. If such an overlay path does not exist, the system transfers the data in Direct Transfer Mode. The system remains in Direct Transfer Mode as long as the performance of the direct connection does not significantly change. The performance of this direct connection is measured for each specified amount of data transfer (16MB in the configuration of the experiments described here). If the system detects the performance of the direct connection has changed significantly (20% from the last measurement in our tests) it starts recomposing the service path. The idea is that a significant change in the performance reflects considerable change in network conditions. When the system observes such a change, it triggers adaptation and recomposes the data transfer path. This recomposition may set up a relay if it finds a suitable overlay node due

to changes in the network condition.

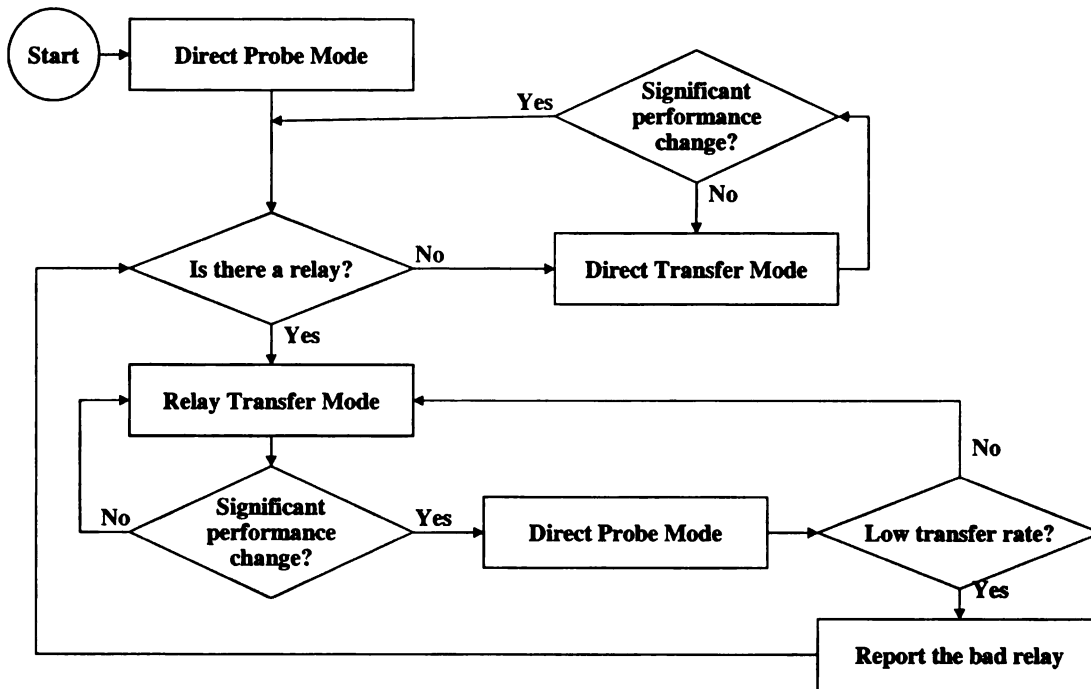


Figure 4.10: Flowchart of the implemented adaptive TCP-Relay algorithm.

If the system does find a suitable overlay node, it establishes an overlay path relayed through this node and transfers the data in the Relay Transfer Mode. The system remains in the Relay Transfer Mode as long as the performance of the relayed path does not change significantly (20% in our tests) or does not become lower (more than 3% in our tests) than the performance measured in the Direct Probe Mode. Otherwise, the system switches to the Direct Probe Mode and measures the performance of the direct connection again. This is necessary, as the condition change may have affected the direct connection performance. After re-measuring the performance of the direct connection, the system re-evaluates the performance of the relay. If the performance of the relay is acceptable according to the recent performance of the direct connection probe, the system continues using the current relay. Otherwise, the system marks the current relay as unsuitable for the corresponding

pair of endpoints. Next, it triggers adaptation to recompose the data transfer path. The overlay node marked as unsuitable, for the particular pair of endpoints, remains an invalid choice for a specified amount of time (300 minutes in our tests, which effectively means till the end of a transfer, since all transfers last less than 300 minutes in these tests). Afterwards, the system removes that node from the list of bad relays. The recomposition relocates the relay function to another overlay node, or switches to Direct Transfer Mode if this time a suitable overlay node is not found.

Empirical Results and Analysis. In this set of experiments, three different strategies are used to transfer data between two endpoints: direct connection, primitive TCP-Relay, and adaptive TCP-Relay. The direct connection strategy is the baseline and uses native TCP/IP connection between two endpoints. In the primitive TCP-Relay strategy, if analysis of the collected inter-node delay information predicts a promising relay node, the system dynamically configures and uses the best promising node as a relay—without measuring the performance of the relay in action (this is the TCP-Relay implementation presented earlier). Finally, the adaptive TCP-Relay strategy performs run-time adaptation of the composed overlay path as described. Table 4.3 shows the list of nodes in this experiment. In the test procedure, the source node (ID 0) sends 128MB of information to 5 destinations (receivers).

Figure 4.11 plots the time taken to transfer 128MB of data from the source (sender) to the destinations. Each data point is the average of at least 3 successful transfers (90% confidence intervals shown). For clarity, we have sorted these plots in ascending order of the direct connection transfer time. These plots show that except for the receiver node 2, the adaptive TCP-Relay performs better than the primitive TCP-Relay. The advantage of the adaptive strategy is particularly prominent for the receiver node 3.

Table 4.3: List of nodes in the adaptive TCP-Relay experiment.

ID	Node Name	Location	
0	planet1.pittsburgh.intel-research.net	USA – PA	Sender
1	planetslug1.cse.ucsc.edu	USA – CA	Receiver
2	planetlab1.cs.cornell.edu	USA – NY	
3	planet2.cs.ucsb.edu	USA - CA	
4	planetlab01.tamu.edu	USA – TX	
5	peace.ri.uni-tuebingen.de	Germany	
6	planetlab1.cs.ubc.ca	Canada	
7	planet-lab-1.csse.monash.edu.au	Australia	
8	planetlab01.cs.washington.edu	USA – WA	
9	plab1.cs.ust.hk	Hong Kong (China)	
10	planetlab01.dis.unina.it	Italy	

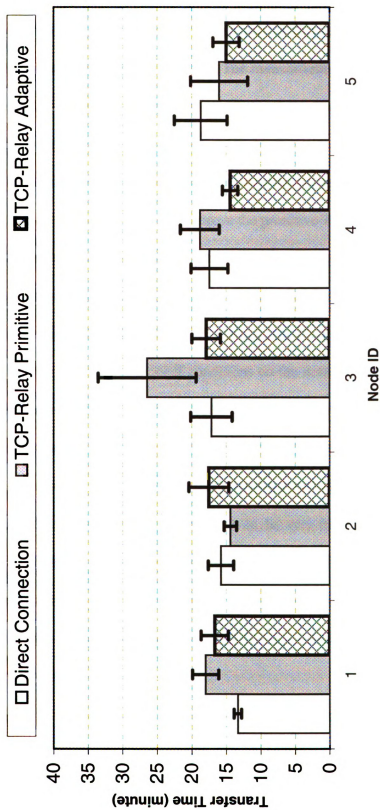


Figure 4.11: Data transfer from planet I.pittsburgh.intel-research.net towards the destinations.

Figure 4.12 plots performance ratio of the primitive and adaptive TCP-Relay services to the direct connection. We observe that the performance of the adaptive TCP-Relay strictly improves as the direct transfer time increases. The main advantage of the adaptive TCP-Relay is avoiding situations where the primitive TCP-Relay performs very poorly (such as the case for receiver 3).

In these tests, the adaptive TCP-Relay service is more *robust* than the primitive one. The adaptive TCP-Relay service will not improve the primitive TCP-Relay performance if the first selected relay node consistently boosts the throughput (in fact, the overhead of the adaptive mechanism may slightly degrade the performance of the TCP-Relay service). But the adaptive service avoids significantly lower performance if a relay node, predicted to be suitable, turns out to operate poorly or if conditions on the network change significantly. We emphasize that this is a rudimentary implementation of the TCP-Relay strategies used to demonstrate operation. The performance of the adaptive TCP-Relay service might be improved via enhanced monitoring and decision-making mechanisms.

In summary, Service Clouds has been shown to be very useful in constructing and deploying the TCP-Relay service. Although relatively simple, the service is capable of providing significant performance improvements, relative to native TCP/IP connections. Next, let us turn to a more complex service, constructing reliable multipath connection in an overlay network.

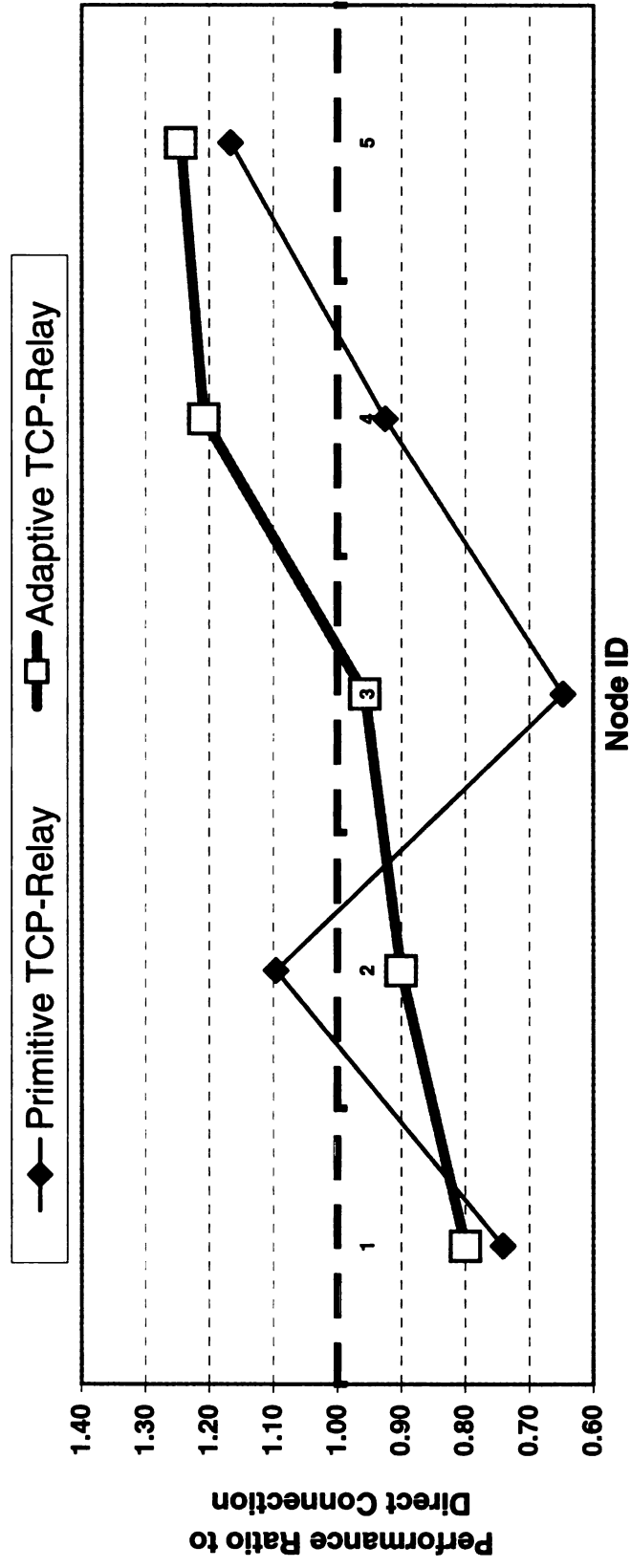


Figure 4.12: Adaptive and Primitive TCP-Relay performance ratios.

4.3 Case Study: Multipath Connections

Establishing multiple connections (e.g., a primary path and one or more backup paths) between a source and destination node can improve communication quality by mitigating packet loss due to network failures and transient delays. This capability is especially important to distributed applications, such as high-reliability conferencing, where uninterrupted communication is essential. However, in overlay networks, selection of high-quality primary and backup paths is a challenging problem due to the sharing of physical links among overlay paths. Such sharing affects many routing metrics, including joint failure probability and link congestion. In a separate work, our research group described TAROM [106], a distributed algorithm for disseminating physical topology information to establish multipath connections. In this case study, we use TAROM in the construction of MCON (Multipath CONnection), a distributed service that automatically finds, establishes, and uses a high-quality secondary path whenever a primary overlay path is established.

4.3.1 Basic Operation

MCON combines traditional distributed multipath computation with topology-aware overlay routing and overlay multipath computation. A topology-aware overlay approach [107] considers shared physical links between two paths as a factor when determining high-quality backup paths. Since sharing of physical links among overlay paths reduces their joint reliability, this method leads to finding more robust backup paths.

The basic operation of MCON connection establishment is depicted in Figure 4.13. To establish a multipath connection, the source node sends a request packet to the destination

node along the primary path (path-establish procedure). This packet collects path composition (physical topology) and quality information (packet loss in this prototype) along the way. Upon reception, the destination node forwards this information to its neighbors in path-explore packets, each of which attempts to find its way to the source (the path-explore procedure). When a node receives a path-explore packet, it uses both the received information and its local state to calculate the joint quality of the primary path and the partial path from itself to the destination. If the joint quality-cost ratio is above a specified threshold, it forwards this information to its neighbors. By adjusting the value of the threshold, the application can control the scope and overhead of this path-explore process. If a node receives path-explore packets from two neighbors, it calculates the joint quality of the two partial paths and forwards the information of the primary and the better partial path to its neighbors. The process terminates when the remaining branches reach the source, providing it with a set of high-quality backup paths.

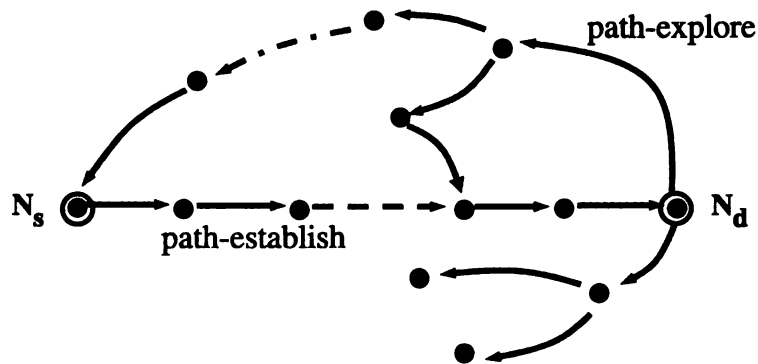


Figure 4.13: Basic operation of MCON.

4.3.2 Implementation Details

In Figure 3.4, the *AMX* interface for MCON requires the caller to simply specify the destination node and, optionally, a resiliency strategy for finding a secondary path. In the current implementation, two resiliency strategies are supported: topology-unaware (TUA) and topology-aware using inference (TA-i). The former attempts to find a high-quality secondary path with respect to the overlay topology, but without considering underlying path sharing in the physical network. The latter uses physical topology information, some of it measured directly and the rest obtained using highly accurate inference techniques [107].

As described earlier, the logic for a distributed service is encapsulated in an overlay engine. In the case of MCON, we had available to us a C++ implementation of the TAROM algorithm, which had been used as part of a simulation study. With only minor modifications, we were able to plug the existing C++ code into the Service Clouds infrastructure as an MCON overlay engine and access it through the Java Native Interface. MCON-specific control services include components to transmit and forward *Path-Establish* and *Path-Explore* packets among nodes, as described earlier. In terms of data services, MCON requires a *Resiliency Adaptor* (an instance of a local adaptor) at both the source and destination. At the source, the adaptor creates a duplicate of the data stream and sends it to the destination over the discovered secondary path. At the destination, the adaptor delivers the first copy of each data packet received and discards redundant copies. In this prototype, we have focused on the overlay service model, thus the data adaptation mechanism deploys a simple adaptor. In general, one may consider complicated data adaptation techniques within the Service Clouds architecture, such as reflection-based MetaSockets [90]. To send

a data stream, after finding a (primary or secondary) path to the destination, the overlay service on the source inserts an application-level header in each outgoing data packet, indicating which overlay nodes are to be traversed on the path toward the destination. The *Source Router* implements a source routing mechanism that routes the data packets through the overlay network along the predetermined path.

Figure 4.14 shows a simplified flowchart of *inquiry packet processing* within Service Clouds. An inquiry packet is processed at multiple stages at each overlay node before it is delivered to the MCON overlay engine. First, if a packet traverses the maximum number of hops and does not reach the destination, it will be dropped. Next, the framework provides information available at the current node. This includes topology of the direct path and packet loss rate toward the forwarder of the received path-explore packet. Finally, the information extracted from a path-explore packet is delivered to the MCON overlay engine.

The overlay engine compares the given partial path quality to other partial paths found so far that exist in its cache. Then, it determines whether the packet under process indicates a better partial path or not. If the packet does not indicate a path with a higher quality, it will be dropped. Otherwise, the framework keeps the inquiry packet in a buffer. This is a one-slot buffer that holds an inquiry packet, rather than propagating it instantly, for a specified time by running a timer thread, which is started whenever an empty buffer is filled. During this time, other packets may be received and indicate a path with a quality higher than the one identified by the current packet in the buffer. In such a case, the framework overwrites the existing packet in the buffer. This mechanism supports a basic idea in MCON computation, which seeks to avoid unnecessary propagation of inquiry packets

in the distributed system. Finally, if the current node is the destination, the framework has found a backup path and the AMX layer will notify the application to begin streaming. Otherwise, the framework forwards the packet (which also includes information about the current node) to all its neighbors except the one that sent the original packet and those that the packet has already traversed (controlled flooding).

4.3.3 Empirical Results and Analysis

In this set of experiments, we selected 20 Planetlab nodes, listed in Table 4.4 (also illustrated on the geographical map in Figure 4.1), and established an overlay network among them (in the form of a complete graph). We chose five sender-receiver pairs from the overlay network and transmitted a UDP stream from each sender to its corresponding receiver using each of three different services: one as direct connection on the Internet without any resiliency services (*NR*), one with topology-unaware multipath service (*TUA*), and one with topology-aware multipath service (*TA-i*).

For the experiments presented here, the application generates data packets of length 1KB and sends them at 30 msec intervals for 3 minutes, which generates 6000 UDP packets for each sample stream. The results presented are the average of seven separate experimental runs.

We compare the performance of TUA and TA-i with NR in terms of packet loss rate reduction, robustness improvement, and overhead delay. The robustness improvement I for a multipath data transmission is defined as the percentage of fatal failure probability (the rate of simultaneous packet losses on all paths) reduction due to the use of a secondary

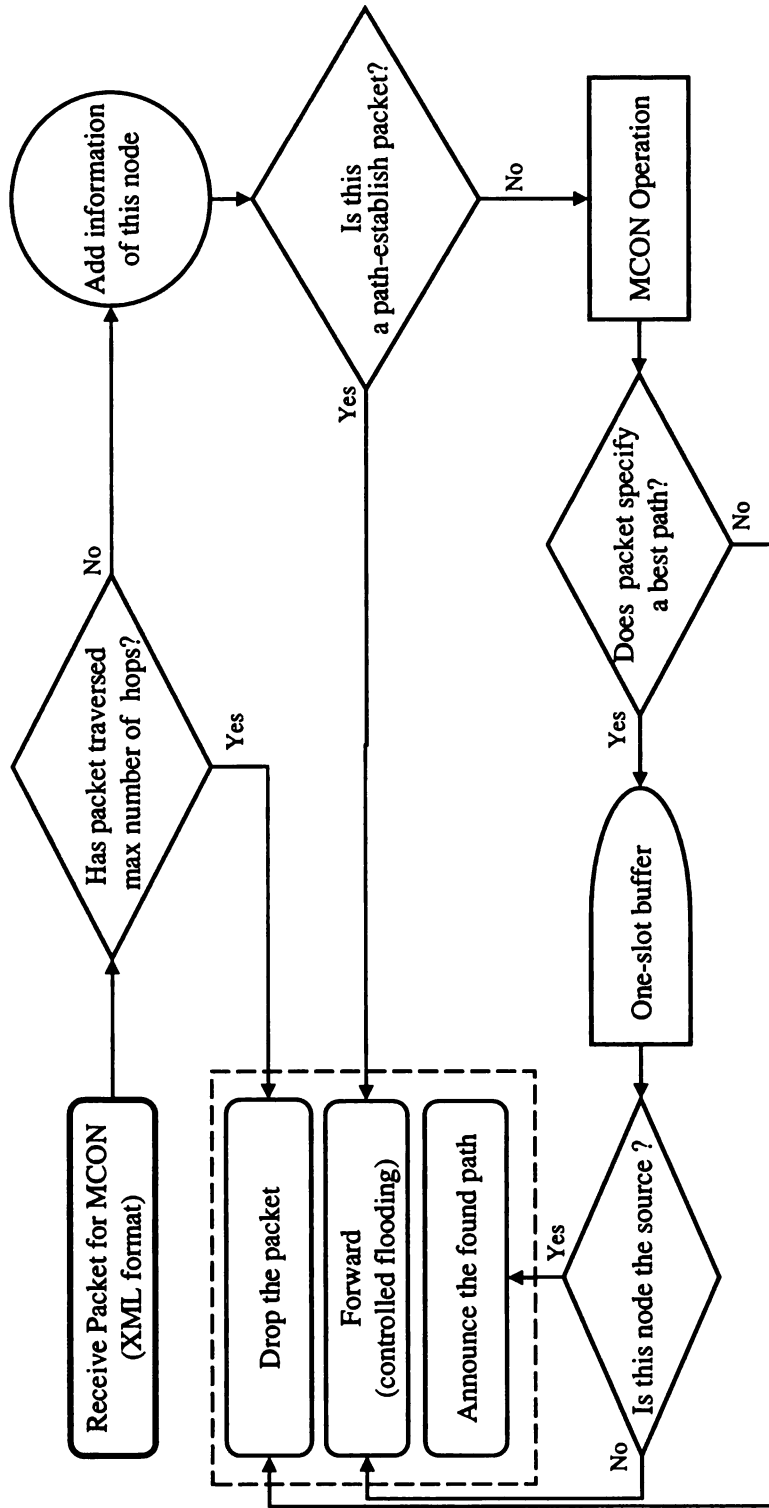


Figure 4.14: MCON inquiry packet processing within Deep Service Clouds.

Table 4.4: List of nodes in the MCON experiment.

ID	Node Name	Location	Sender/ Receiver	Pair #
0	planetlab1.enel.ucalgary.ca	Canada	Sender	1
1	plab2.ee.ucla.edu	US – CA	Receiver	
2	planetlab-3.cs.princeton.edu	US – NJ	Sender	2
3	planet-lab-2.csse.monash.edu.au	Australia	Receiver	
4	planetlab2.sics.se	Sweden	Sender	3
5	planetlab2.arizona-gigapop.net	US – AZ	Receiver	
6	planetlab2.win.trilabs.ca	Canada	Sender	4
7	planetlab1.cs.uit.no	Norway	Receiver	
8	csplanetlab2.kaist.ac.kr	South Korea	Sender	5
9	blast.cs.uwaterloo.ca	Canada	Receiver	
10	pli1-pa-4.hpl.hp.com	US – CA		
11	planetlab1.nbgisp.com	US – OR		
12	planetlab3.comet.columbia.edu	US – NY		
13	planetlab-2.eecs.cwru.edu	US – OH		
14	planet03.csc.ncsu.edu	US – NC		
15	planetlabone.ccs.neu.edu	US – MA		
16	planetlab1.een.orst.edu	US – OR		
17	planetlab1.rutgers.edu	US – NJ		
18	planetlab1.netlab.uky.edu	US – KY		
19	planetlab3.uvic.ca	Canada		

path:

$$I = \frac{F_s - F_d}{F_s},$$

where F_s is the single path failure probability of the primary path, and F_d is the double path failure probability of both the primary and the secondary path.

Figure 4.15 shows the results for different test groups (aggregated according to sender-receiver pairs) and the overall results. Figure 4.15(a) demonstrates that compared with NR, both TUA and TA-i can substantially reduce the average packet loss rate. Figure 4.15(b) shows TA-i exhibits higher robustness improvement than TUA, except in group 3. For group 4, the average robustness improvement in TA-i is over 20 times greater than that of TUA. Figure 4.15(c) plots the dynamics of packet loss rate of each test. The figure shows packet loss rate is very low in most tests. However, there are packet loss bursts in some tests. By using multipath routing, especially TA-i, Service Clouds can reduce both the intensity and frequency of such loss bursts. Again, TA-i exhibits better performance than TUA. Figure 4.15(d) plots the CDF (cumulative distribution function) of packet loss rate in all tests. We observe both TUA and TA-i can reduce the frequency of transmissions with relatively high packet loss rate, and TA-i is statistically better than TUA (since most TA-i data points are above and to the left of the TUA data points).

Finally, while the use of the Service Clouds infrastructure introduces additional computational and communication overhead, Figures 4.15(e) and 4.15(f) show that the effect on data packet delay is relatively small when compared to NR. Figure 4.15(e) plots the average packet delay in each test group. The figure shows that except in group 1, Service Clouds approaches (TUA, TA-i) usually do not significantly increase packet delay. Figure 4.15(f)

plots the CDF of packet delay in all tests. The figure shows, the extra delay introduced by Service Clouds is tolerable in most tests. However, some test exhibits doubled delay for TUA and TA-i. This may not be acceptable to some time-sensitive applications. Such unexpected delays may be the result of sudden system slowdown—for example, if it becomes overloaded by other applications running on the same node—that delays data packet processing (for routing or delivery).

These results demonstrate that the MCON service, implemented within the Service Clouds prototype, can improve communication reliability in real Internet environments. Moreover, we emphasize that this improvement exists, even though the implementation is not optimized (we simply plugged in an existing algorithm and implemented rudimentary monitoring routines).

Further, we note that the experiments described here were conducted under “normal” network conditions. The true value of the MCON service is to be demonstrated in situations where serious failures occur along the primary path, and a high-quality secondary path continues to deliver the data stream to the destination.

4.4 Related Work

The idea of splitting TCP connections to improve performance, also known as *TCP spoofing*, originally received attention as a means of improving the performance of Internet traffic over satellite links [108, 109]. Unlike satellite applications where the relay location is fixed in advance, the key issues in TCP-Relay are how to construct a system that can find promising relay nodes for an individual data transfer and create the correspond-

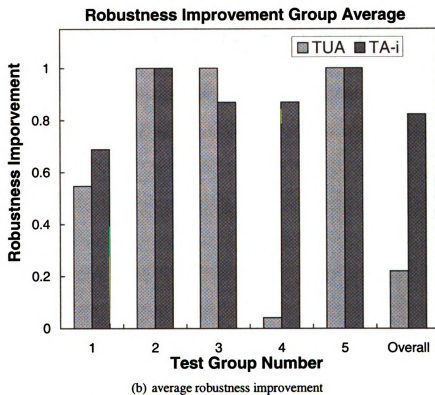
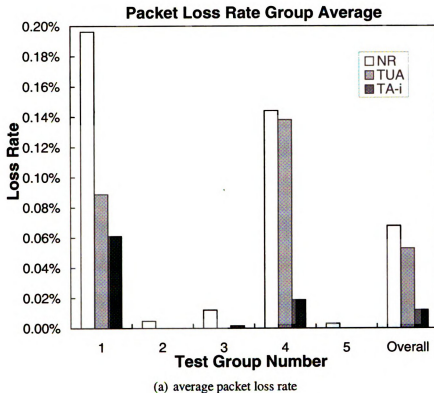
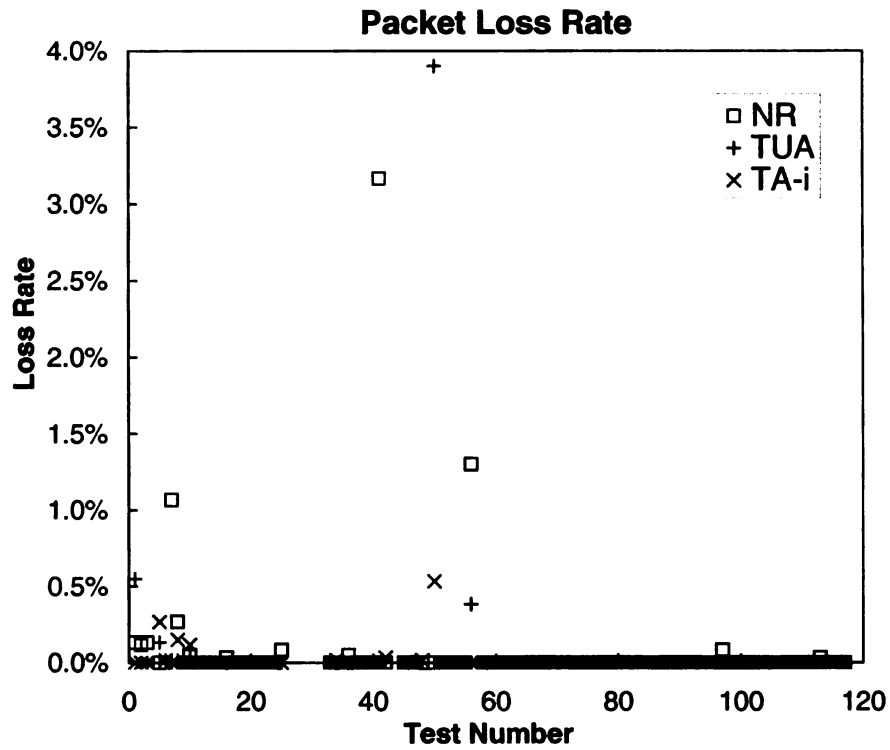
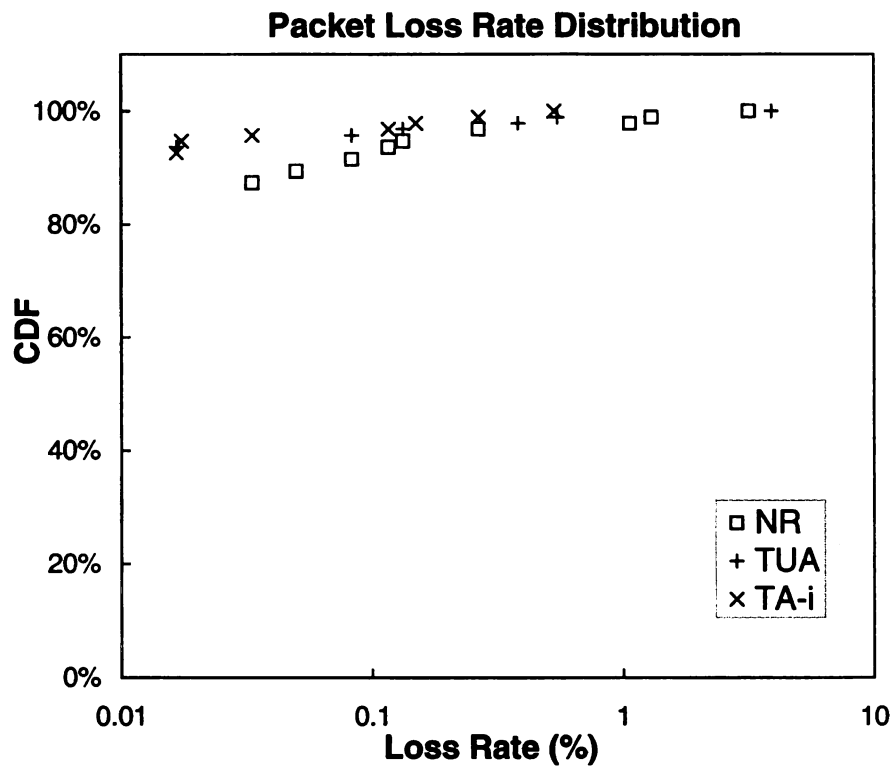


Figure 4.15: MCON empirical results on PlanetLab (continued on the next page).

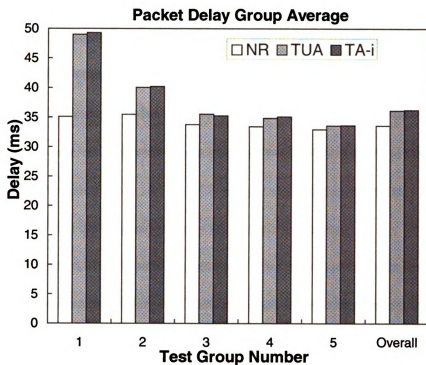


(c) packet loss rate dynamics

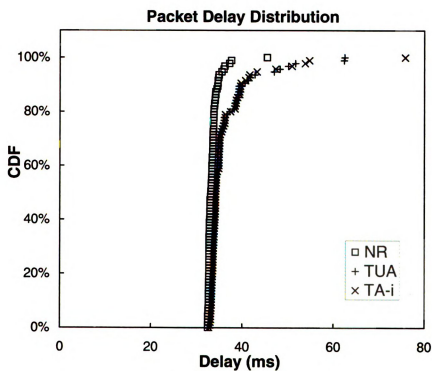


(d) packet loss rate CDF

Figure 4.15: MCON empirical results on PlanetLab (continued on the next page).



(e) average packet delay



(f) packet delay CDF

Figure 4.15: MCON empirical results on PlanetLab.

ing service on demand. We note that TCP relays can be more easily deployed than some other approaches to improving TCP throughput, such as using advanced congestion control protocols (e.g., XCP [110], HS-TCP [111], FAST[112]), which require either router support or kernel modifications. Alternatively, TCP relays can be used in tandem with such techniques. However, as such solutions require network-layer support in routers [110] or kernel-level modifications [111, 112], TCP relays organized in an overlay network may offer a more practical deployment strategy for many applications.

The basic problem in multipath computation is to select a set of paths, under cost constraints, that optimize a particular set of metrics. Goals include minimizing the joint failure probability (considered by the MCON computation), maximizing aggregate throughput, or minimizing packet delay. A variety of algorithms have been proposed to solve this problem in physical networks [113–118]. Despite the effectiveness of existing algorithms for such environments, fundamentally different approaches are needed in overlay networks, where link sharing among overlay paths can affect path joint quality. Several overlay multipath approaches have been proposed recently [119–123]. However, they mainly focus on resource management rather than path interactions. In the case study presented earlier, MCON exploits physical topology information to achieve higher accuracy in the calculation of joint failure probability; it calculates joint path quality while considering link sharing, and it terminates path exploration if the cost-benefit ratio is too high. Network-layer topology information is so useful to overlay network applications that several methods have been developed to exploit it for efficient overlay network construction and management [107, 124–126]. MCON includes an assessment of how the availability of such information affects the performance of multipath computation. It provides a topology exchange switch for each

primary path, which can be turned on and off according to a flag in the probe to reduce communication cost in case the topology information is locally available.

4.5 Summary

In this chapter, we presented the use of Service Clouds to develop overlay-based services on the Internet. We showed how the Service Clouds architecture accommodates components to support distributed overlay algorithms. In the first case study, we deployed dynamic TCP relays that expedite data transfer between two nodes. We described in detail how the Service Clouds framework implements the TCP-Relay service operation. Empirical results on the PlanetLab testbed demonstrate the effectiveness of the framework in creating relays dynamically with minimal overhead, and transferring data at a higher speed in many cases. In the second case study, we studied establishing a high-quality overlay path as a shadow to the direct Internet connection to support resilient streaming. We used the MCON overlay algorithm, which exploits physical topology of the network to find backup paths for the primary path with minimal joint failure probability. Experiments on the PlanetLab show the effectiveness of the framework for the MCON algorithm and provide robust streaming.

We emphasize that Service Clouds is a general model and architecture. Further, the prototype provides a toolkit that supports rapid deployment of services for real testbeds. Actually, deployment of MCON on PlanetLab was our pilot work to investigate and design an infrastructure for autonomic communication services. Using the partial framework from the MCON experience, we designed and deployed the TCP-Relay service in less than two weeks, which experience itself helped us to redefine the model and create a more complete

framework.

Chapter 5

Mobile Service Clouds: Extending Service Clouds to the Wireless Edge

In this chapter, we introduce the *Mobile Service Clouds* model, which extends Service Clouds to the wireless edge of the Internet. We define the wireless edge as the set of nodes that are, at most, a few hops away from the wired infrastructure. In this model, a collection of overlay nodes close to the wireless edge form a service cloud that supports autonomic services in mobile computing. We discuss services required at the wireless edge and show how the proposed model supports mobile computing. Specifically, we address the following issues: first, a model to support pervasive mobile computing based on Service Clouds; second, adaptation of multimedia data streaming towards wireless clients; third, self-managing behavior to compose resilient service paths; finally, deployment and evaluation of services to support robust streaming at the wireless edge.

We use the Mobile Service Clouds prototype to perform three case studies. In the first study, we investigate the cross-layer cooperation between the middleware and the operating

system to establish transient proxy services for high-quality video streaming at the wireless edge. In the second study, we conduct an experiment in which we assess the ability of the Mobile Service Clouds approach to establish transient proxies for mobile devices, monitor the service path, and support dynamic reconfiguration (with minimal interruption) when the proxy node fails. In the third study, we demonstrate how the proposed model supports high-quality, pervasive streaming for mobile users moving about at the wireless edge.

In terms of horizontal cooperation, which is the focus of the second case study, a service path comprises nodes within deep clouds and mobile clouds, where proxy services are established to support specific needs at the wireless edge, for example, forward error correction, which is more efficient to be close to the wireless edge to avoid unnecessary traffic on the wired segment, resulting from parity packets. Further, as mobile clients join, leave, and move about different wireless network domains, proxies need to be created, reconfigured, or terminated to support pervasive services. This functionality is realized through collaboration among deep and mobile clouds. On the other hand, vertical cooperation between the middleware and the operating system is more practical on wireless clients and edge nodes, since access to the operating system kernel can be made possible by an administrator to support local services (in deep service clouds, nodes are usually providing services for users outside of a domain, and administrators may not be willing to allow access to the operating system). This is especially useful in efficient data adaptation, as we show in the first case study.

We note that extensive research has been conducted in the design of proxy services to support data transcoding, handle frequent disconnections, and enhance the quality of wireless connections through techniques such as forward error correction [127–131]. Rather

than addressing the operation of specific proxy services, in this work we concentrate on the dynamic instantiation and reconfiguration of proxy services, made possible by the Service Clouds infrastructure, in response to changing conditions.

This chapter is organized as follows. Section 5.1 investigates services at the wireless edge and introduces the model. Section 5.2 introduces the prototype instantiation for the Mobile Service Clouds case studies. Sections 5.3, 5.4, and 5.5, respectively, describe three case studies and present empirical results. Section ?? discusses related work. Finally, Section 5.6 concludes this chapter.

5.1 Mobile Service Clouds Model

Figure 5.1 depicts an extension of the Service Clouds concept to support mobile computing. In this model, mobile service clouds comprise collections of overlay hosts that implement services close to the wireless edge, while deep service clouds perform services using an Internet overlay network (such as the PlanetLab wired hosts used in our earlier study). In a manner reminiscent of Domain Name Service, this *federation* of clouds cooperates to meet the needs of client applications. Typically, a mobile service cloud will comprise nodes on a single intranet, for example, a collection of nodes on a university campus or used by an Internet Service Provider (ISP) to enhance quality of service at wireless hotspots. A mobile user may interact with different mobile service clouds as he/she moves about the wireless edge, with services instantiated and reconfigured dynamically to meet changing needs.

Figure 5.1 shows an example in which a mobile user is receiving a live or interactive video stream on his mobile device. Service elements are instantiated at different locations

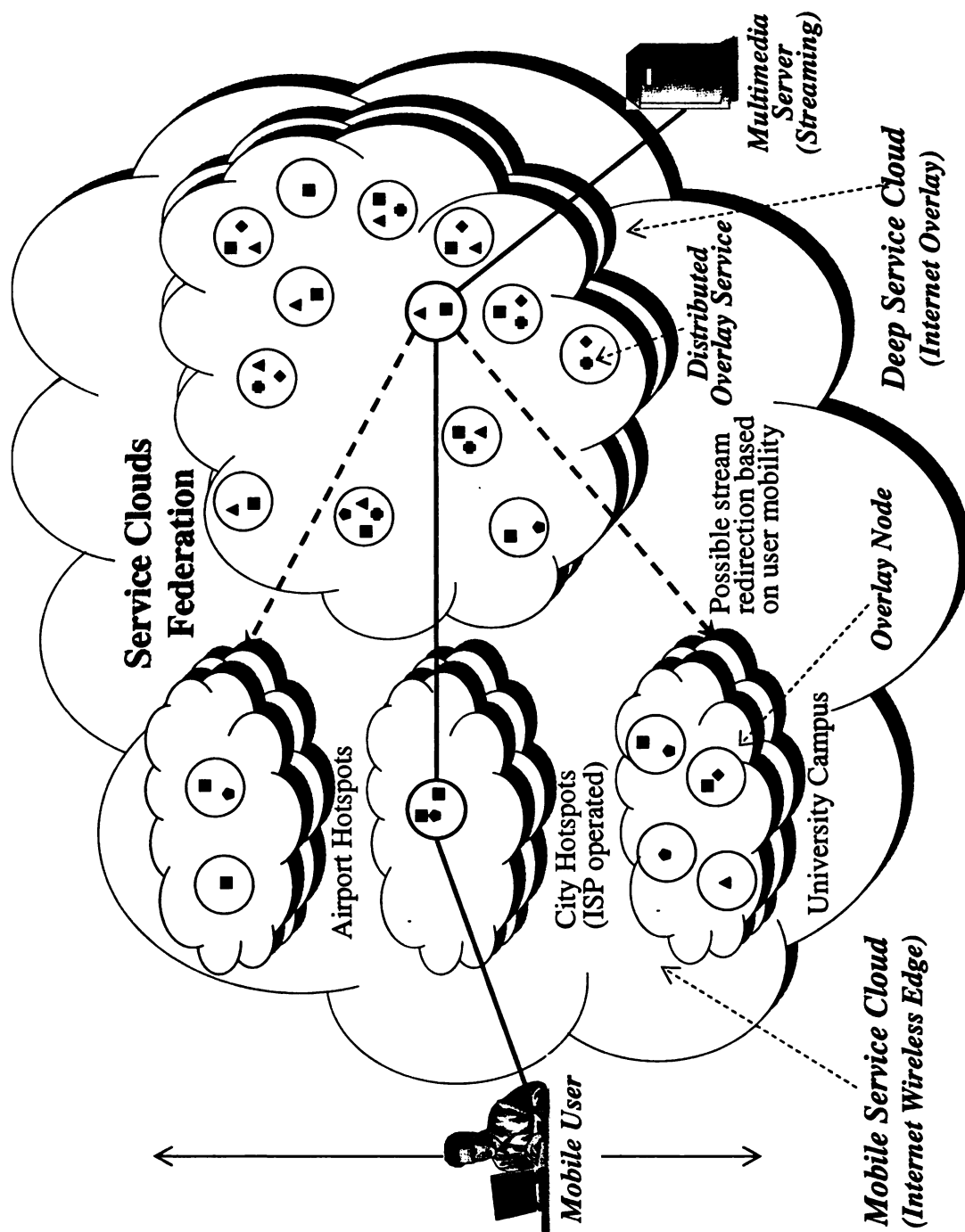


Figure 5.1: Example scenario involving Mobile Service Clouds.

along an overlay path according to application requirements and current conditions. For example, if the user is connected via a relatively low-bandwidth wireless link, a video transcoder may be established close to the video source, to reduce the bit rate on the video stream and avoid wasted bandwidth consumption along the wired segment of the path. On the other hand, a proxy service that uses FEC and limited retransmissions to mitigate wireless packet losses may be established on a mobile service cloud node at the wireless edge. The operation of the proxy service depends on the type of data stream to be transmitted across the wireless channel. Over the past few years, our research group has investigated proxy services for reliable multicasting [132], audio streaming [133, 134] and video streaming [133, 135].

As the user moves within an area serviced by a single mobile service cloud, or among different service clouds, the proxy can be migrated so as to move along with the user. Such a service is an instance of a *transient* proxy [28], introduced in Chapter 3. There are several reasons to keep the proxy close (in terms of network latency) to the mobile user:

- *Quality of service of the data stream delivery.* Low latency enables more complex types of error control. For example, EPR [135] is a forward error correction method for video streaming in which the proxy encodes video frames using a block-erasure code, producing a set of parity packets. The proxy sends a subset of the parity packets “pro-actively” with the stream. Additional parity packets are sent in response to feedback from the mobile device to handle temporary spikes in loss rate. However, the effectiveness of these additional parity packets depends on the round-trip delay between the mobile device and the proxy; if the delay is too long, the parity packets

arrive too late for real-time video playback.

- *Resource consumption.* A proxy that implements forward error correction increases the bandwidth consumption of the data stream. When the mobile device connects to a different Internet access provider, if the proxy is not relocated, then the additional traffic may traverse several network links across Internet service providers. While the effect of a single data stream may be small, the combined traffic pattern generated by a large number of mobile users may have a noticeable effect on performance.
- *Policy of the service provider.* While an ISP may be willing to use its computational resources to meet the needs of users connected through its own access points, this rule may not apply to mobile hosts using access points belonging to another provider. In the same way that the mobile device changes its access point but remains connected, the proxy services on the connection may need to move in order for the new connection to be comparable to the old one.

In the case studies presented in this chapter, we address two concerns in the operation of proxies at the wireless edge. First, cross-layer interaction between the middleware and the operating system to create transient proxy services for data stream adaptation. Second, reconfiguration of proxies to provide highly available services. We also consider fault tolerance when a proxy service fails; if a proxy service crashes or becomes disconnected, another instantiation of the service should assume its duties with minimal disruption to the communication. Moreover, we use dynamic proxy services to provide pervasive, high-quality streaming for mobile users.

5.2 Prototype Implementation

For the prototype implementation of Mobile Service Clouds, we introduced new components but also reused several others. Figure 5.2 provides a detailed view of the Service Clouds prototype, showing those components introduced or used in the Mobile Service Clouds case studies (unshaded boxes), as well as those used only in the Deep Service Clouds case studies (shaded boxes) from Chapter 4.

The “Coordination and Service Composition” manages the interaction between a mobile host and a service cloud federation. Tasks include selection of a node in a deep service cloud—using the *Service Path Composition overlay engine*—called the *primary proxy*, which coordinates composition and maintenance of the service path between two end nodes. The Service Path Computation engine also finds a suitable node in a mobile service cloud on which to deploy the transient proxy services (FEC in our studies). We also require lower-level services to use in identifying potential proxies. The *RTT Meter* component uses “ping” to measure round-trip time (RTT) to an arbitrary node, and the *Path RTT* component measures end-to-end RTT between two nodes whose communication is required to pass through an intermediate node.

The *Service Gateway* component implements a simple protocol to accept and reply to service requests. Upon receiving a request, it invokes the overlay engine to find a suitable primary proxy. The *Service Composer* component implements mechanisms for composing a service path. It uses the *Relay Manager* to instantiate and configure a UDP relay on the primary proxy and the transient proxy. The UDP relay on the transient proxy enables the infrastructure to intercept the stream and augment it with FEC encoding. Accordingly, as

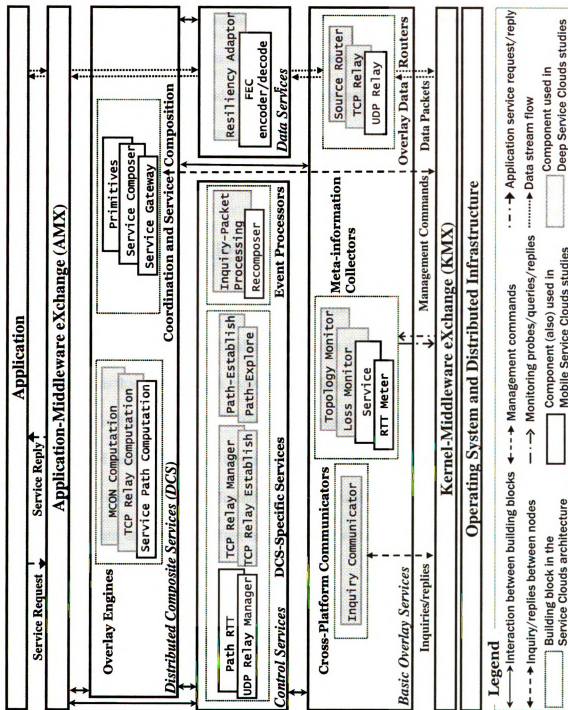


Figure 5.2: Instantiation of the prototype for Mobile Service Clouds.

soon as the FEC proxy service is instantiated, the *Service Monitor* on the transient proxy begins sending heartbeat messages through a TCP channel toward the service monitor on the primary proxy. The *Recomposer* component on the primary proxy tracks the activity of the service monitors. Upon detecting a failure, it starts a self-healing operation that recomposes the service path and restores communication. Additionally, a monitor on client middleware notifies a proxy service of changes, such as low-battery status or change of IP address on a mobile client, and the recomposer reconfigures the service as necessary.

5.3 Case Study: High-Quality Wireless Video Streaming

Mobile computing environments exhibit the operating conditions that differ greatly from their wired counterparts. In particular, applications must tolerate the highly dynamic channel conditions that arise as users move about. Moreover, the computing devices being used by different end users may vary in terms of display characteristics, processor speed, memory size, and battery lifetime. Given their synchronous and interactive nature, real-time applications such as video conferencing are particularly sensitive to these differences. In particular, data adaptation is often necessary to overcome high packet loss and resource limitations at the wireless edge.

5.3.1 Basic Operation and Experimental Setup

Figure 5.3 shows the experimental testbed, a hybrid network that combines wired and wireless network segments. In our experiments, a video stream is multicasted over a local area network and is routed to a group of laptop computers operating in ad hoc mode. Wireless

communication is prone to packet loss. FEC filters are inserted into the transient proxies as needed to improve the multimedia QoS for the wireless nodes. Specifically, we use block erasure codes [136], which compensate packet loss by sending a number of parity packets. These codes are especially beneficial in multicast streaming when the link layer retransmissions do not exist and packets are lost at multiple nodes independently. In the scenario presented here, we consider just one path in the multicast tree to show the effectiveness of the cooperation model between the middleware and the operating system called *KMX* (Kernel Middleware eXchange). Whenever an ad hoc node experiences intolerable packet loss, the video stream is intercepted and passed to the transient proxy at the Router node. The middleware proxies on the Router and Receiver nodes apply FEC encoding/decoding on the video stream transparently to the video application. In general, using transient proxies, adaptations such as FEC can be applied only to those parts of the network that need them.

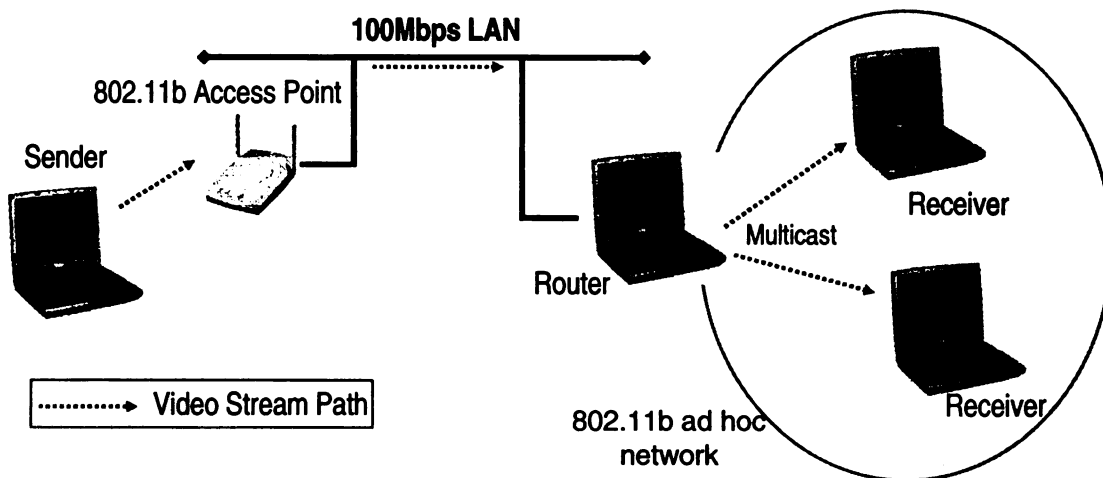


Figure 5.3: The experimental testbed for high-quality wireless video streaming.

The testbed comprises two Mobile Pentium 4 laptops with 1GB RAM (Router and Receiver nodes) and one Mobile Pentium III laptop with 256MB RAM (Sender node).

The transmission power of the network adapter card on the **Router** (Cisco 340) has been reduced to 1mW to make it possible to conveniently perform the tests over short distances. The data rate over the ad hoc network is 2Mbps. The **Sender** and **Receiver** run Windows XP Professional, and the **Router** runs Linux Red Hat 9.

For video streaming and playback we used a video application developed at our laboratory, enabling us to instrument it for various measurements such as number of packets dropped due to delay. However, we emphasize that the prototype framework is general and off-the-shelf commercial applications could be used. The video clip is the first 55 seconds of a standard reference video [137]; its specifications are given in Table 5.1. Multicast packets are 1KB at the application-level.

The routing subsystem at the **Router** node is originally configured to route the multicast packets, sent by the video application on the **Sender**, towards the **Receivers** in the wireless segment of the network. When packet loss in the wireless segment of the network increases, KMX can use “iptables” [138] to configure the Linux kernel on the **Router** to intercept the multicast packets sent by the **Sender** and redirect them to a transient proxy. Using TRAP/J, we have incorporated MetaSockets into the UDP Relay (Figure 5.2) so that FEC can be added dynamically (we explain the details in the following section). In these experiments, the transient proxy on the **Router** breaks each packet into two packets and performs FEC (4 , 2) encoding. Specifically, each packet is split into two packets, then four FEC encoded packets are created by the FEC encoder filter. Only two out of four encoded packets are needed at the **Receiver** for the FEC decoder filter to generate the original packet.

Table 5.1: Video specifications in the high-quality wireless video streaming experiment.

Video	Highway drive
Number of Frames	1375 (original highway drive contains 2001)
Frame Rate	25 fps
Playing Time	55 seconds
Encoding	Mpeg-4 avi Using DivX Pro 5.0.5 Codec 200 kbps encoding rate Max key frame interval: 12 frames
Video Size	352*288 (CIF)
File Size	1386 KB (on hard disk)
Streaming	1 KB packet-size (including a 48-byte application-layer header) 1905 packets totally

5.3.2 Implementation Details

We have built a rudimentary KMX framework for this case study on Linux, where most operating system services reside in the kernel. The prototype shows how the KMX model can be implemented using adaptive middleware technologies from the RAPIDware project [98] and off-the-shelf facilities available in Linux.

We use MetaSockets [90] to construct transient proxies for mobile environments. The composition and behavior of a MetaSocket can be adapted at run time in response to changing conditions. For example, a loss-detector filter can monitor the packet loss and another filter can perform forward error correction (FEC) encoding upon detection of intolerable packet loss. Figure 5.4 shows the MetaSendMSocket architecture. MetaSendMSocket is a meta-level component over the regular multicast socket that enables adaptation of a stream that is sent through the socket. A similar meta-level component enables the configuration of filters in the receiver multicast socket corresponding to the filters at the sender socket. Details of the MetaSocket mechanism can be found in [90]. Further, we use TRAP [139]

to weave the MetaSockets into the middleware components to provide Data Services (Figure 5.2).

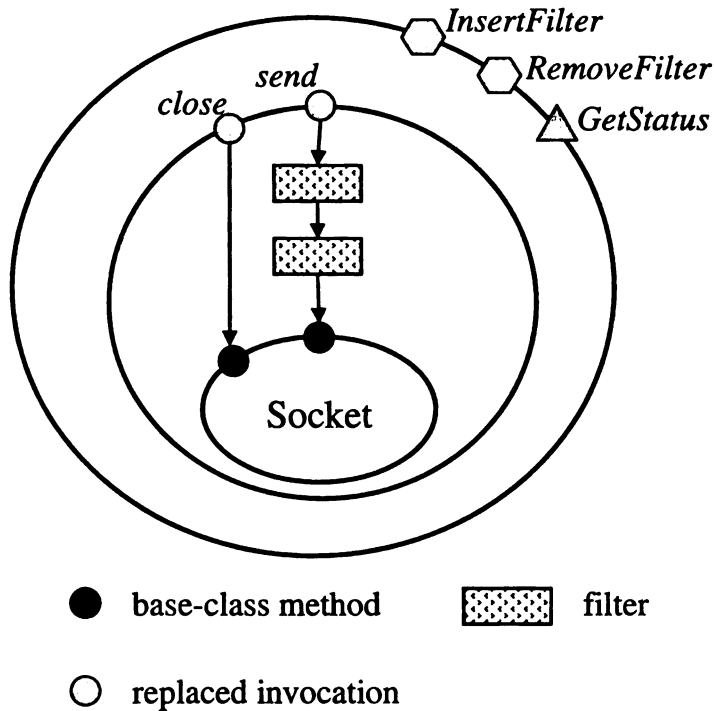


Figure 5.4: MetaSendMSocket architecture [140].

Figure 5.5 shows a high-level representation of the TRAP/J operation, a Java implementation of TRAP, a generator that takes as input a Java application and a list of classes that are to be made adaptable. TRAP/J uses the AspectJ compiler to generate an *adapt-ready* version of the application. Specifically, an aspect is generated for each of the specified classes, providing a hook that can be used to introduce new behavior at run time. In our case study, we use TRAP/J to enable run-time reconfiguration of transient proxies.

Finally, the prototype uses off-the-shelf tools to configure the Linux kernel for routing and intercepting multicast packets. The SMCRoute [141] tool enables configuration of multicast routing in the Linux kernel. Specifically, a software component can use SMCRoute to customize multicast routes in the kernel routing subsystem. In addition, the

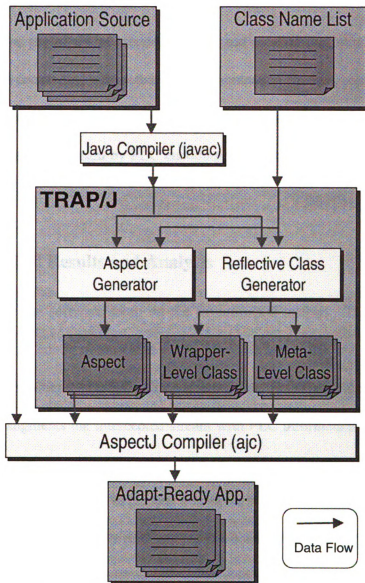


Figure 5.5: TRAP/J operation at compile time [139].

Linux kernel supports packet manipulation by a framework called Netfilter. The iptables facility [138] in Linux can be used to configure Netfilter to intercept data streams that traverse the kernel routing subsystem. Specifically, a software component can use iptables to define rules for intercepting packets that pass through the kernel and meet a specific set of criteria (such as protocol type, sender address, and receiver port number). In our case study, when high packet loss is detected in a communication channel, packets belonging to a specified data stream arriving at a node can be intercepted and passed up to a transient proxy, which adapts the stream by FEC encoding the packets before they are sent to the next node.

5.3.3 Empirical Results and Analysis

Figure 5.6 shows the software setup on the experimental testbed. This setup supports streaming in the two configurations that we described earlier. In the first configuration, the Router forwards multicast packets without interception. In the second configuration, the transient proxy augments the intercepted stream with FEC information. We demonstrate the effect of video stream adaptation when an ad hoc node is experiencing high packet loss. Figure 5.7 shows the percentage of packets that arrive at a Receiver in the two configurations. This figure plots the average of 6 runs for two configurations. FEC introduced by the transient proxy improves the overall packet reception rate from 72.9% to 91.4%. Figure 5.8 shows snapshots of the video frames at the indicated play times. These still pictures show how adaptation of the data stream can noticeably improve the visual quality of the video.

Since the percentage of received packets and a few samples of still pictures do not

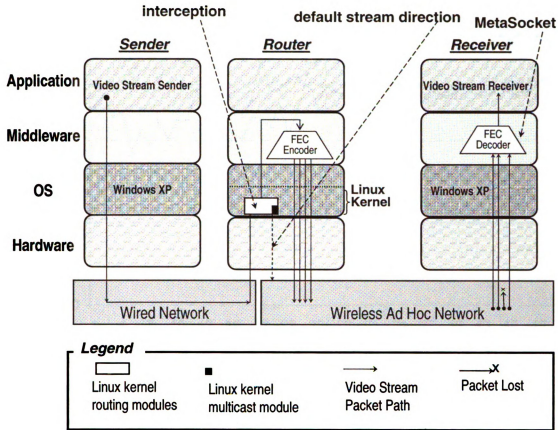


Figure 5.6: Software setup on the experimental testbed.

directly reflect the effect on the video quality, we also use VQM [142] to evaluate the quality of the received video. VQM facilitates off-line perception-based estimation of video quality, in addition to the traditional peak signal-to-noise ratio measurement. We used the video-conferencing model in VQM to measure the quality of the received video. This model is optimized for video-conferencing applications with bit rates from 10 kbps to 1.5 Mbps (the encoding rate of the video in this case study is 200 kbps).

Figure 5.9 shows the quality of the received video measured by VQM in the two configurations described above. We configure VQM to break 52 seconds (1300 frames) of the video into six 12-second segments, such that each segment overlaps the last 4 seconds of the previous segment. Each segment is scored between 0 to 5, where 5 indicates

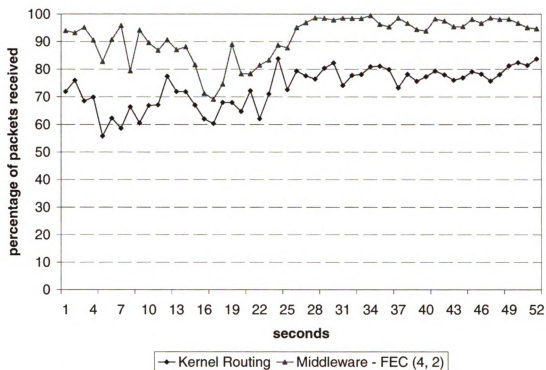


Figure 5.7: Percentage of packets received at the receiver in the high-quality wireless streaming experiment.

imperceptible impairment in the received video. Table 5.2 shows the overall packet loss and the corresponding quality estimation in the two configurations for the entire video (55 seconds). Adaptation by the transient proxy improves the VQM score from 2.13 to 3.94 (a 84.97% boost), which shows a significant improvement in the quality of the received video.

Table 5.2: Overall packet loss and quality in the high-quality wireless video streaming experiment.

	Kernel Routing	Middleware FEC (4, 2)
Total Packet Loss (middleware layer)	27.01%	8.6%
Root Cause Analysis [Blurring] / [Jerky Motion] / [Block Distortion]	[2.5%] / [45%] / [73.5%]	[0%] / [32%] / [59.83%]
VQM Score (video conferencing model)	2.13	3.94



(a) Kernel Routing (0:00:17.89)



(b) Middleware FEC (0:00:17.89)



(c) Kernel Routing (0:00:36.34)



(d) Middleware FEC (0:00:36.34)

Figure 5.8: Video snapshots.

As demonstrated by the above results, intercepting and applying FEC filters at the intermediary nodes can significantly improve quality of the received video. Another possible strategy for the above scenario is to perform end-to-end FEC encoding, with FEC encoding at the wired sender and FEC decoding at the receivers. However, such an approach consumes more resources and generates more traffic without providing a QoS better than the presented approach. This possible strategy is not efficient because virtually all the packet loss occurs in the wireless segment of the network. Further, KMX interception can be applied selectively to streams experiencing poor conditions, and the error control can be

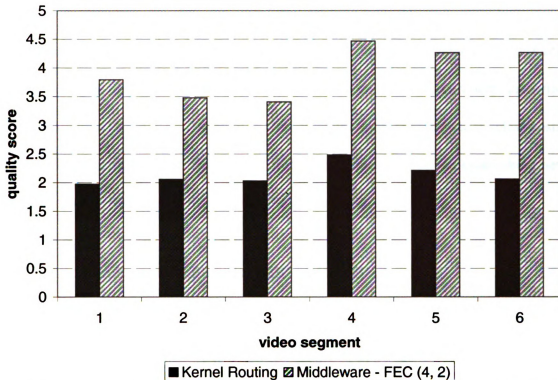


Figure 5.9: Video quality measurement in the high-quality wireless video streaming experiment.

implemented in a manner best suited to the data type.

Another issue in live multimedia streaming is delay, especially in interactive applications such as video conferencing. In such real-time applications, if packets reach the destination later than expected, the video player at the application layer will drop them. The delay introduced by the FEC encoding and decoding depends on the processing power and load of the intermediary nodes. If FEC processing cannot be performed fast enough, some packets may miss their playback deadline at the receiver. This situation can degrade, rather than improve, the quality of the received video. Therefore, in general, it may also be necessary to adapt the stream in other ways at the sender, for example, by changing the video encoding bit rate, in addition to FEC adaptation at the intermediary nodes.

5.4 Case Study: Reconfigurable Proxies

In the second case study, we applied the prototype to the problem of dynamically instantiating and migrating proxy services for mobile hosts. We conducted experiments involving data streaming across a combination of PlanetLab nodes, local proxies, and wireless hosts. This study demonstrates the effectiveness of the model and prototype in establishing new proxies and migrating their functionality in response to proxy failures.

5.4.1 Basic Operation and Experimental Setup

In this scenario, the user of a mobile node on a wireless link wants to receive a multimedia stream (e.g., in an interactive video conference or in a live video broadcast), despite changes in network connections and failures of hardware and software components. In this case, Mobile Service Clouds needs to fulfill the following requirements: (a) the quality of the received stream must remain acceptable as the wireless link experiences packet loss; (b) stream delivery should not be interrupted as conditions on the service path change (e.g., when user movement causes a wireless network domain change, or when a service node fails); (c) the video stream must be transcoded to satisfy resource restrictions such as wireless bandwidth and processing power at the mobile device. In our implementation for this case study, we address the first two of these requirements; transducing has been studied extensively [143–146].

Figure 5.10 shows the configuration used in the experiment. It comprises four PlanetLab nodes in a deep service cloud and two workstations on our department intranet in a mobile service cloud, listed in Table 5.3. These systems are all Unix/Linux-based machines.

We have used a PlanetLab node to run a UDP streaming program and a Windows XP laptop to receive the stream over a wireless link. The middleware software on the mobile client connects to a Service Gateway node ($N1$) and requests the desired service. Gateway nodes are the entry point to the Service Clouds: they accept requests for connection to the Service Clouds and designate a service coordinator based on the requested service. In this work, we assume that gateway nodes are known in advance, as with local DNS servers. Other methods, such as directory services, can be integrated into the infrastructure to enable automatic discovery of gateway nodes. Upon receiving the request, the gateway begins a process to find a node to act as the primary proxy ($N4$), and when completed, informs the mobile client of the selection. The primary proxy receives details of the desired service, sets up a service path, and coordinates monitoring and automatic reconfiguration of the service path during the communication.

Table 5.3: List of nodes in the experiment of self-managing services at the wireless edge.

M	senslap10.cse.msu.edu	Mobile User
S	planetlab1.cs.cornell.edu	UDP Streamer
N1	planet-lab-1.csse.monash.edu.au	Service Gateway
N2	planetlab01.cs.washington.edu	Candidate Primary Proxy
N3	planetlab1.cs.ucsb.edu	Candidate Primary Proxy
N4	planetlab1.csail.mit.edu	Candidate Primary Proxy (chosen at run time)
W1	countbasey.cse.msu.edu	Wireless edge relay
W2	arctic.cse.msu.edu	Wireless edge backup relay

A gateway might consider several factors in deciding on a node as the primary proxy for a requested service: security policies of the client and service cloud components, round-trip time between the node and the communication endpoints, and the computational load at the node. In this example, $N4$ is chosen such that the overlay service path round-trip

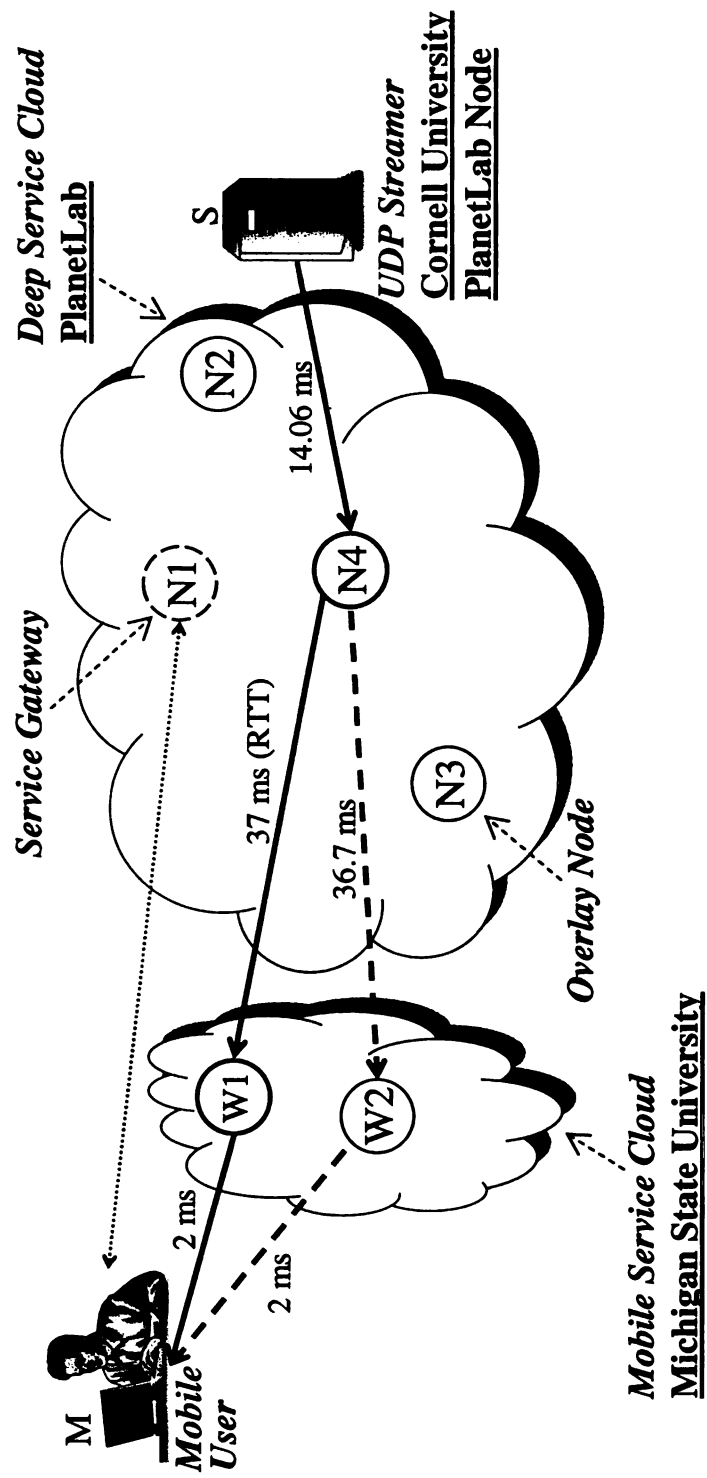


Figure 5.10: The experimental setup for self-managing services at the wireless edge.

time between the two endpoints is minimal.

Next, the infrastructure instantiates FEC functionality at the transient proxy ($W1$) in a mobile service cloud and dynamically re-instantiates the service on another node when the $W1$ fails. “Failure” can be defined in different ways: high computational load, high RTT to the client due to change in access point used by the client, software failure of the service, or hardware failure. To study and test a basic self-repair in Service Clouds, we simply inject a failure by terminating the service process on $W1$.

In the example depicted in Figure 5.10, the Service Clouds client middleware, residing on the laptop, sends a service request to the gateway node $N1$, which chooses $N4$ as the primary proxy and informs the client. Next, the client software sends a primary proxy service request to $N4$, which constructs an overlay service path comprising a UDP relay on itself and a UDP relay augmented with an FEC encoder on $W1$. As soon as $W1$ starts the service, it begins sending heartbeat beacons over a TCP connection to $N4$ that indicate the service node is active. $N4$ runs a monitoring thread that listens for beacons from $W1$ (sent every 5msec in our experiment). If it detects a failure, that is, receives no beacons from $W1$ for a specified period (100 msec in our experiment), it reconfigures the overlay service path to use another node in the mobile service cloud.

5.4.2 Empirical Results and Analysis

To test dynamic reconfiguration of a service path, the program running on $W1$ is terminated. We evaluated two different strategies to realize self-healing at the time of failure detection: (1) *on demand backup* where another node, $W2$, is configured dynamically as

soon as failure is detected; (2) *ready backup* where $W2$ is configured as a backup at the same time of the $W1$ configuration, so the system only needs to configure the relay on $N4$ to forward the stream to $W2$ instead of $W1$.

We have measured the percentage of packets received at the wireless node. Figure 5.11 plots the average of 12 runs for 50 millisecond epochs, indicating the situation when $W1$ fails and system recovers automatically (in this experiment, the program on $W1$ stops functioning after relaying 500 packets). As the plot shows, the system can completely recover from the failure at $W1$ in less than 0.4 seconds. The “on demand backup” is slightly slower, since system has to instantiate and configure the proxy service. In the ready backup case, the service is instantiated at the time of service composition, yielding a faster response.

In our continuing studies, we are investigating dynamic creation and migration of proxies to support seamless multicast streaming when mobile users change network domains. We are also studying middleware framework to enable autonomic client interaction with the infrastructure.

5.5 Case Study: Seamless Mobility

The previous section provided a basic case study on the operation of a reconfigurable proxy service. In this section, we present a more complicated case study that shows how the Service Clouds infrastructure supports seamless, high-quality streaming in highly dynamic situations and relocation of users at the wireless edge. The particular scenario we consider is a multicast-capable proxy service that also supports mobility. Here, a set of clients wish to receive a multimedia stream (e.g, in video-conferencing or live video broadcast), and the

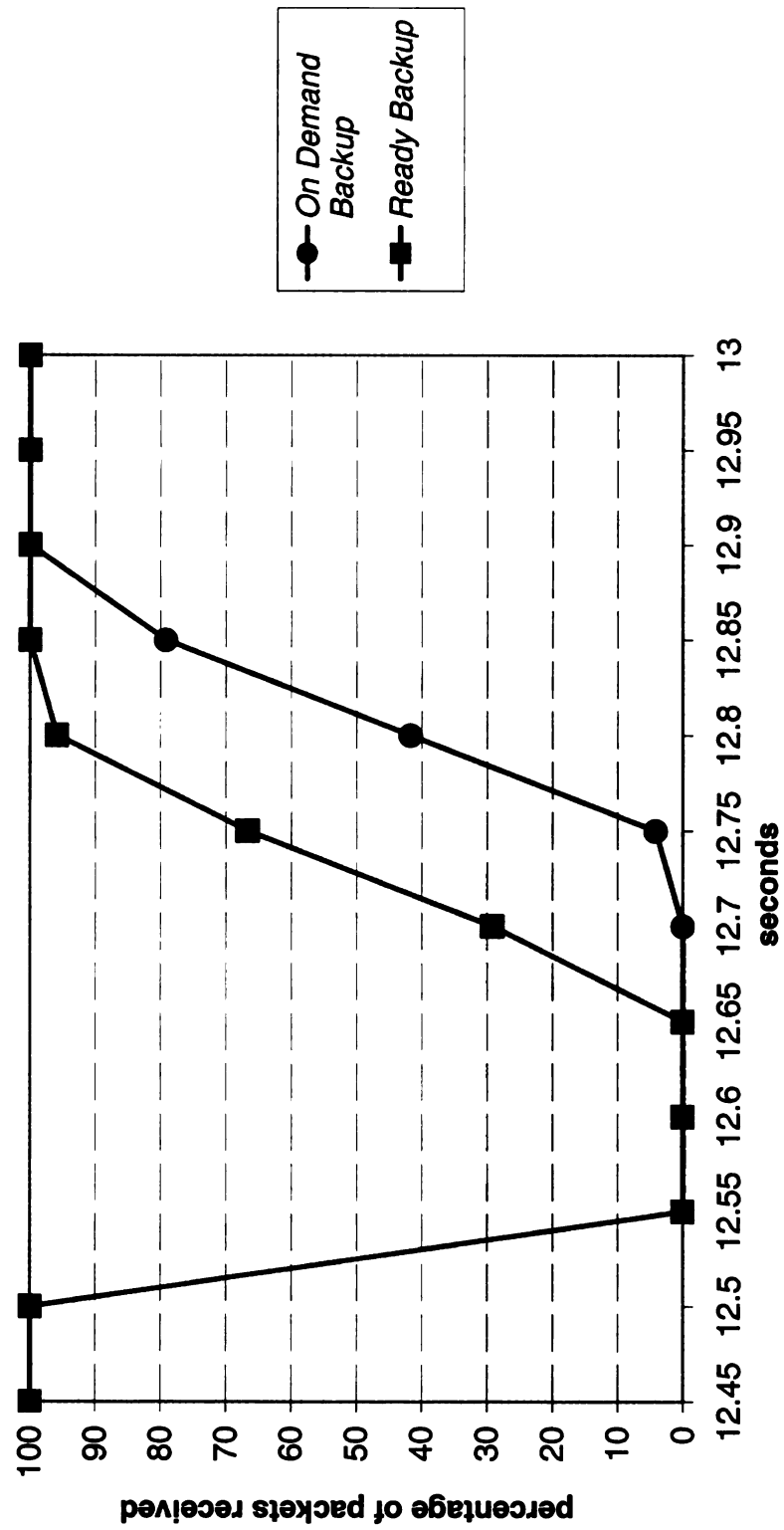


Figure 5.11: Packet loss during proxy failure at the wireless edge.

infrastructure fulfills the following requirements: first, avoiding stream interruptions as a user relocates and connects to a new network domain, gaining a new IP address; second, providing a high-quality reception regardless of the network connection.

5.5.1 Basic Operation and Experimental Setup

Figure 5.12 shows the testbed, with 3 PlanetLab nodes in a deep service cloud, two workstations in a mobile service cloud on our university intranet, and two laptops with RTP-based video players (listed in Table 5.4). Subnet A is a wired LAN and subnet B is wireless. The middleware software on a client connects to a Service Gateway ($N3$) and requests the video. The Primary Proxy ($N1$) creates a service path and coordinates monitoring and automatic reconfiguration during the communication. In this scenario, proxies at the wireless edge deploy two functionalities: multicasting and forward error correction (FEC). Since IP multicasting is not commonly available on the Internet, the stream is unicast toward the wireless edge, where the proxy multicasts it toward the clients. To maintain the quality of the video, the proxy applies FEC on the stream when a client detects high packet loss rate. The infrastructure supports continuous streaming by dynamic instantiation of proxies, while users roam along different subnets.

Figure 5.12 depicts the following scenario:

1. User $M1$ on wired subnet A requests to watch the video from the server; accordingly, the primary proxy creates a service path comprising streaming relays on $N1$ and a unicast-to-multicast proxy on $E1$.
2. User $M2$ requests the same video on wired subnet A; since the video is already being

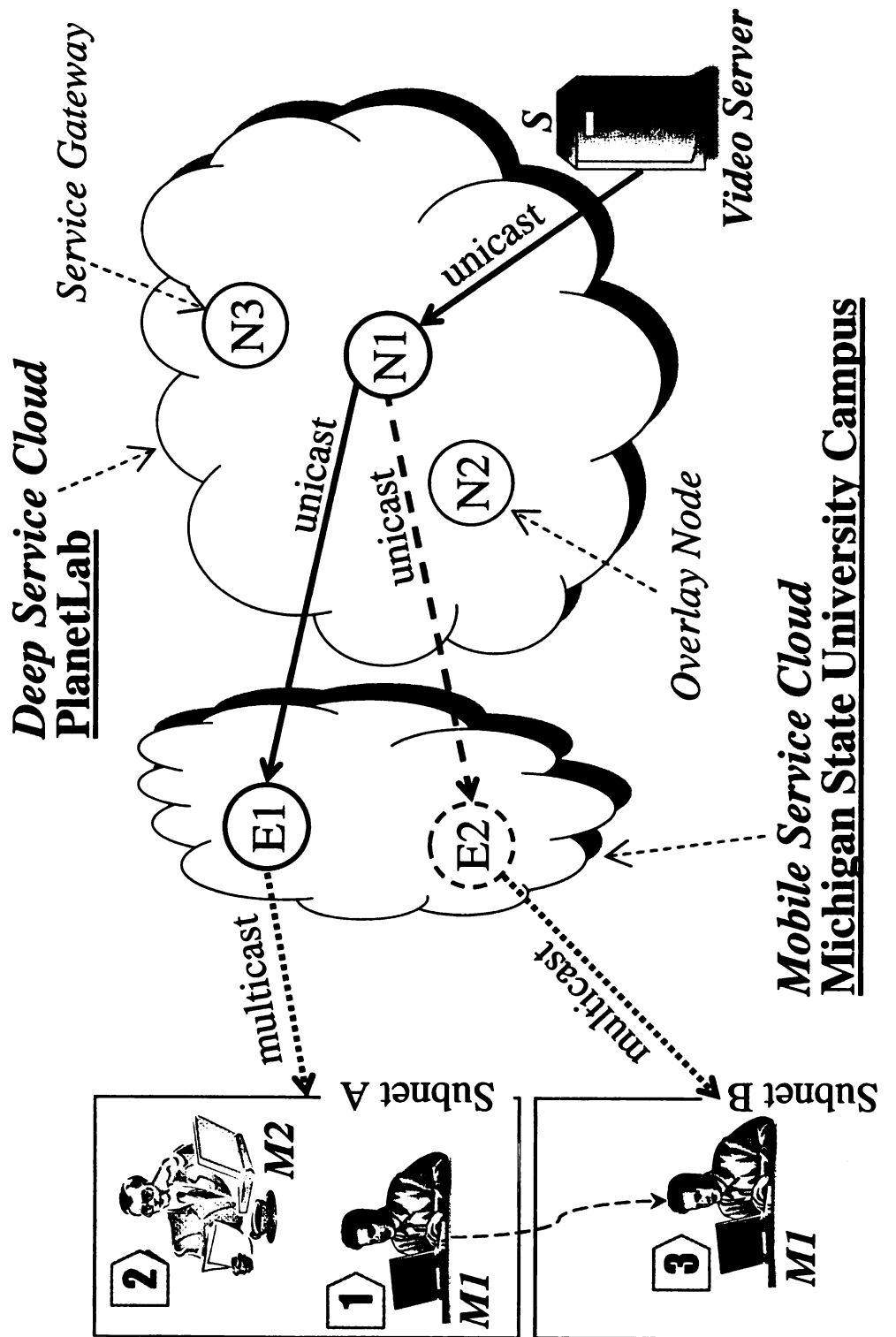


Figure 5.12: The experimental setup for seamless mobility.

Table 5.4: List of nodes in the seamless mobility experiment.

M1	senslap10.cse.msu.edu	User 1 (mobile)
M2	senslap12.cse.msu.edu	User 2
S	copland.cse.msu.edu	Video Streaming Server
N1	planetlab1.cs.unibo.it	Candidate Primary Proxy
N2	planetlug1.cse.ucsc.edu	Candidate Primary Proxy (chosen at run time)
N3	planetlab3.uvic.ca	Service Gateway
E1	countbasey.cse.msu.edu	Wireless edge node
E2	senslap11.egr.msu.edu	Wireless edge node

multicast in subnet A, no extra configuration is necessary, except registering user *M2* as a service receiver and configuring *M2* to receive the video.

3. User *M1* switches from wired connection on subnet A to wireless connection on subnet B; the middleware on the client detects change of IP address and notifies the primary proxy. At this point, the primary proxy extends the service path to make the stream available in subnet B via a proxy on *E2*. Moreover, since the connection to *E2* is wireless, if packet loss rate becomes intolerable, the proxy performs FEC encoding.

5.5.2 Empirical Results and Analysis

In this implementation, the stream comprises audio and video packets sent over separate UDP sockets. For test purposes we stream a prerecorded 30fps motion-JPEG video. Whenever the client in the wireless subnet detects intolerable loss rate (20%), the proxy

in the mobile cloud FEC encodes the stream by breaking each packet to four packets and sending them along with four extra parity packets. Figure 5.13 plots the packet loss rates for audio at *M1*. We have deployed FEC encoding only on the audio stream to balance QoS and bandwidth consumption. High-quality wireless streaming may require more complicated adaptation techniques to transcode the video, but our focus here is on the software infrastructure, rather than transcoding methods. As the plot shows, 5 seconds into the trace the user switches from wired subnet to the wireless one and the network loss rate raises significantly. At time 11, the loss rate exceeds 20%. Accordingly, the system enables FEC encoding, based on the feedback from the client middleware, and effectively mitigates the packet loss rate observed by the application.

In Section 5.4, we described the use of Service Clouds to construct self-healing proxy services. The tests described above demonstrate that the Service Clouds infrastructure can also be used to construct adaptive service paths at the wireless edge. The infrastructure provides self-management in the dynamic instantiation and migration of proxy services. The resulting system can adapt to dynamic changes in the environment of mobile nodes, including changes in connectivity and channel conditions, while maintaining a high quality of service on the stream delivered to the user.

5.6 Summary

In this chapter, we addressed the issue of dynamic services at the wireless edge of the Internet. We introduced *Mobile Service Clouds*, an infrastructure based on the Service Clouds model, for the deployment of services to support mobile computing. The model supports

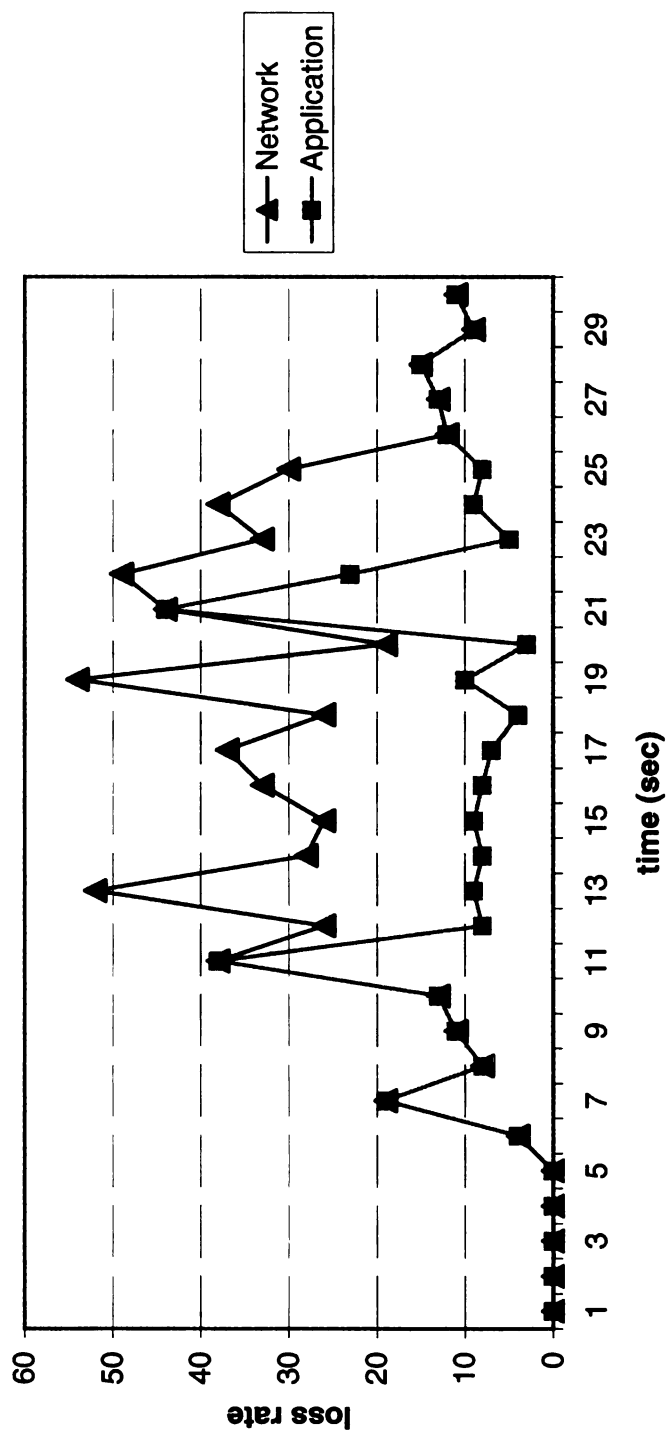


Figure 5.13: Audio packet loss rate at mobile node M1.

dynamic creation, reconfiguration, and migration of services to provide QoS services for mobile users. This adaptive behavior is especially important at the wireless edge, where conditions are highly dynamic, users relocate, and proxy services are more effective when located close to the clients. We conducted case studies where proxy services are dynamically established and configured at the wireless edge. First, we showed how cross-layer cooperation between the middleware and the operating system layers supports data adaptation to achieve QoS streaming at the wireless edge. Next, we used the Mobile Service Clouds prototype to support proxy instantiation and fault tolerance on a testbed comprising PlanetLab nodes and hosts on a university intranet. Third, we demonstrated how dynamic instantiation and functionality migration of proxies at the wireless edge support pervasive mobile computing. Results demonstrate the usefulness of the model and the effectiveness of the prototype in providing adaptive services to mobile clients.

Chapter 6

Dynamis: Dynamic Service Composition for Distributed Stream Processing

Recent advances in sensor technologies and wireless networks can provide organizations and individuals with enormous amounts of data about the physical environment [147, 148]. Sensor networks are important to a number of emerging civilian and military applications, including computational ecology, environmental monitoring, surveillance and security, smart structures, critical infrastructure protection, and battlefield awareness. Once collected, sensor readings can be assembled into data streams and transmitted over computer networks for processing and storage. Processing the data as it is collected is needed to meet the needs of real-time applications as well as to facilitate later searching and analysis of repositories. Without timely processing, the sheer volume of the data might preclude the extraction of information of interest. While the components of some data processing services are relatively static, others depend on dynamics in the user population and their queries.

Realizing dynamically composable services requires a communication infrastructure that can adapt dynamically to changing conditions and user requirements, and establish and reconfigure complex services on demand. As we have already demonstrated, achieving this goal has been facilitated by the advent of *overlay networks* [67]. With increasing computing capacity and connectivity of nodes on the Internet, researchers and developers have begun to use overlay networks as a chassis to support complex services [19, 23, 76, 149]. To provide robust services, an overlay infrastructure needs to integrate several technologies, including dynamic software configuration techniques to adapt services in the presence of varying conditions [150] and monitoring systems to detect failures and changes in the execution environment [151].

In this chapter, we focus on the problem of mapping distributed services onto an overlay network. This functionality is an integral part of any overlay-based streaming framework and typically requires a probing mechanism to locate suitable nodes on which to instantiate new data processing operators [152], or to reconfigure and possibly share existing operators [153]. The probing protocol should incur minimal traffic overhead while producing a high-quality mapping of services onto the overlay infrastructure. The quality can be measured in terms of metrics such as end-to-end delay, load balance, security, and cost.

The contributions of this study are threefold. First, we propose *distributed selection*, an optimization technique that supports the design of efficient probing mechanisms. We demonstrate that applying distributed selection to probing algorithms can significantly reduce probing overhead. Second, we introduce an extensible algorithm based on distributed selection, called *Dynamis*, to realize efficient probing for overlay service composition. Third, we report results of an experimental study on the PlanetLab Internet testbed, where

we assess the performance of Dynamis and other service composition algorithms.

This chapter is organized as follows. Section 6.1 formulates the problem of service composition and probing. Section 6.2 describes the Dynamis approach in general. Section 6.3 presents the details of the Dynamis algorithm. Section 6.4 gives results of the experimental investigation. Section 6.5 discusses related work. Finally, Section 6.6 concludes this chapter.

6.1 Problem Formulation

In this section, we provide basic definitions and formally state the probing problem for distributed service composition.

Service Element (Operator). A service element $S = \{F, R, I, O\}$ is a service entity executing on a single node, where F specifies the set of functions carried out by the service; R defines the resource requirements of the service, for example, {memory, processing power, output bandwidth}; I specifies acceptable input, for example, {bit rate, resolution} in a video stream; and O states the generated output specifications, for example, {size, fps} in a video stream.

Service Path. A service path P is an alternating directed sequence $P : n_0, l_0, n_1, l_1, \dots, n_n, l_n (n > 0)$ of overlay nodes and overlay links l_i , where $l_i = (n_{i-1}, n_i)$, such that each node executes one or more service elements (S_i) each time it is visited on the sequence.¹ Figure 6.1 depicts an example service path S consisting of three service

¹ In graph theory, such a traversal is called a *walk*, and a *path* is a walk in which no vertex is repeated. Since in the service composition literature the term *service path* is common and used loosely, we also use this term.

elements (S_1, S_2, S_3) distributed on overlay nodes between two endpoints. We note that it is necessary to specify *both* nodes and links in the path, since an overlay network may be multichanneled, with multiple overlay links following different physical paths between the same two nodes [154].

Service Graph. A service graph $\lambda = \cup\{P_i\}$ comprises the union of one or more connected service paths (P_i). Figure 6.1 shows an example of a service graph that forms a multicast tree between a source and destinations.

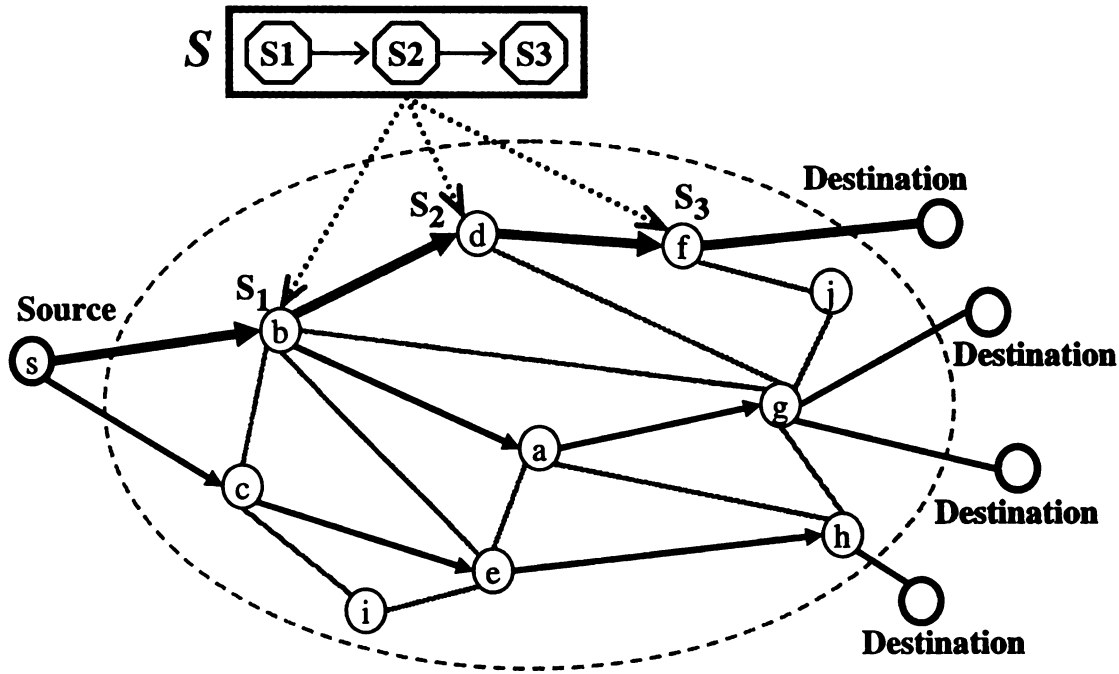


Figure 6.1: Service graph example (highlighting a service path).

Service Graph Quality. The quality of a service graph is the end-to-end quality observed at the endpoints. The quality measurement is domain specific and may include overlay stretch properties such as end-to-end delay and packet loss, or non-functional aspects such as security, reliability, and cost.

Hosting. We say that a node can host a service element if that node has available

resources to execute a service element and satisfy domain-specific criteria of the service graph, such as reliability, security, and end-to-end delay.

Comparable Service Graphs. Two service graphs are comparable if they map the same functionality onto the overlay network. In Figure 6.2, the service paths at the bottom of the figure, $a - g$ and $c - e - h$, both host service elements S_1 , S_2 , and S_3 , so they are comparable; they are not comparable to the $d - f$ graph, which hosts only services S_2 and S_3 .

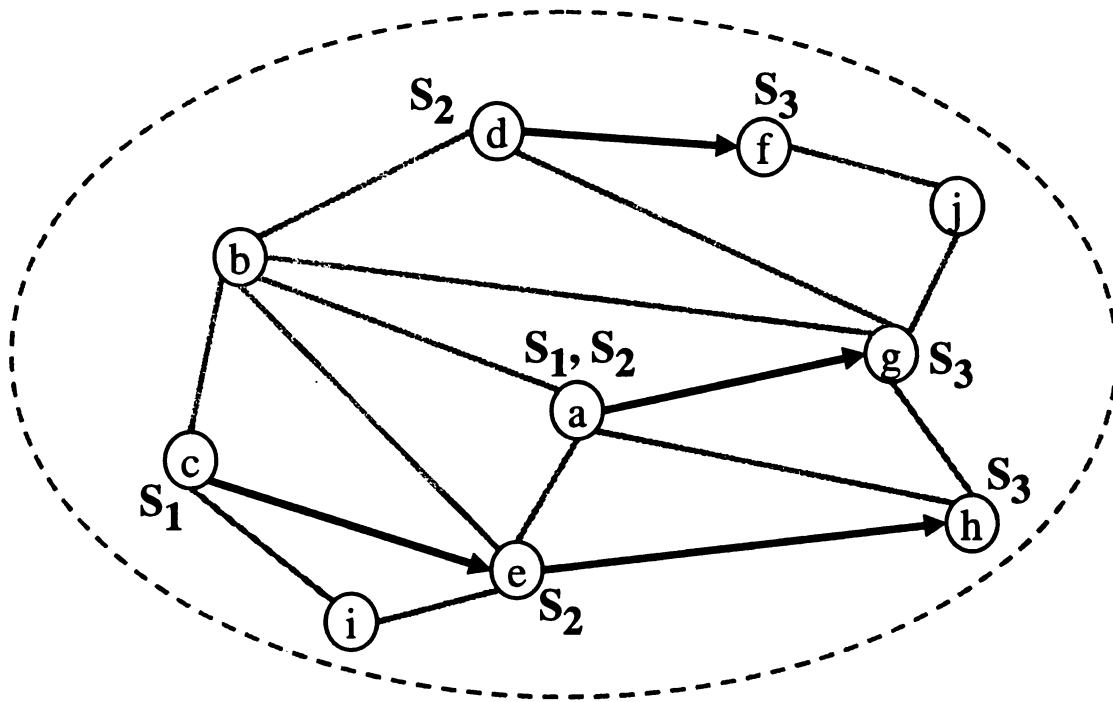


Figure 6.2: Examples of service graphs.

In this work we focus on the special case of composing service *paths*. Figure 6.3 shows examples of service paths established in an overlay network. In these figures, the notation $\{S_i, \dots, S_j\}$ represents an unordered set of service elements, that is, functionality of members is commutative. The notation (S_i, \dots, S_j) represents an ordered set of service elements, that is, the functionality of each member depends on the previous one and is

non-commutative. We note that in S_C , element S_3 is not actually executed twice; rather, the specification states that S_4 and S_5 depend on S_3 . If the same service element needs to be executed more than once, then each instance has to be labeled differently.

Formally, we can state the problem addressed here as follows: Given an overlay graph G of n nodes and a service specification S , find a path P in G such that P can host S .

6.2 Service Composition Framework

This section first describes a framework for composing services and mapping them to an overlay network. Next, it introduces the Dynamis probing technique to support distributed service composition with minimal probing overhead.

Framework. Figure 6.4 shows the overall operation model of the service composition in an overlay infrastructure for autonomic communication, such as Service Clouds. In this model, when an application requests a service, the infrastructure composes an overlay service graph that implements the requested service. The composition substrate has four main parts: overlay engine, probing protocol, service composer, and monitor. The overlay engine and probing protocol provide the overlay algorithm that sends multiple probes to find service graph candidates and select a composition mapping onto the overlay network. A probe describes specifications of a service path including constraints on ordering of service elements, resource requirements for each service element, and expected end-to-end service path quality. The overlay engine contains the metrics to quantify quality of service graphs. It selects suitable overlay nodes to form a service graph and passes the selected mapping to the service composer. The service composer dynamically instantiates or con-

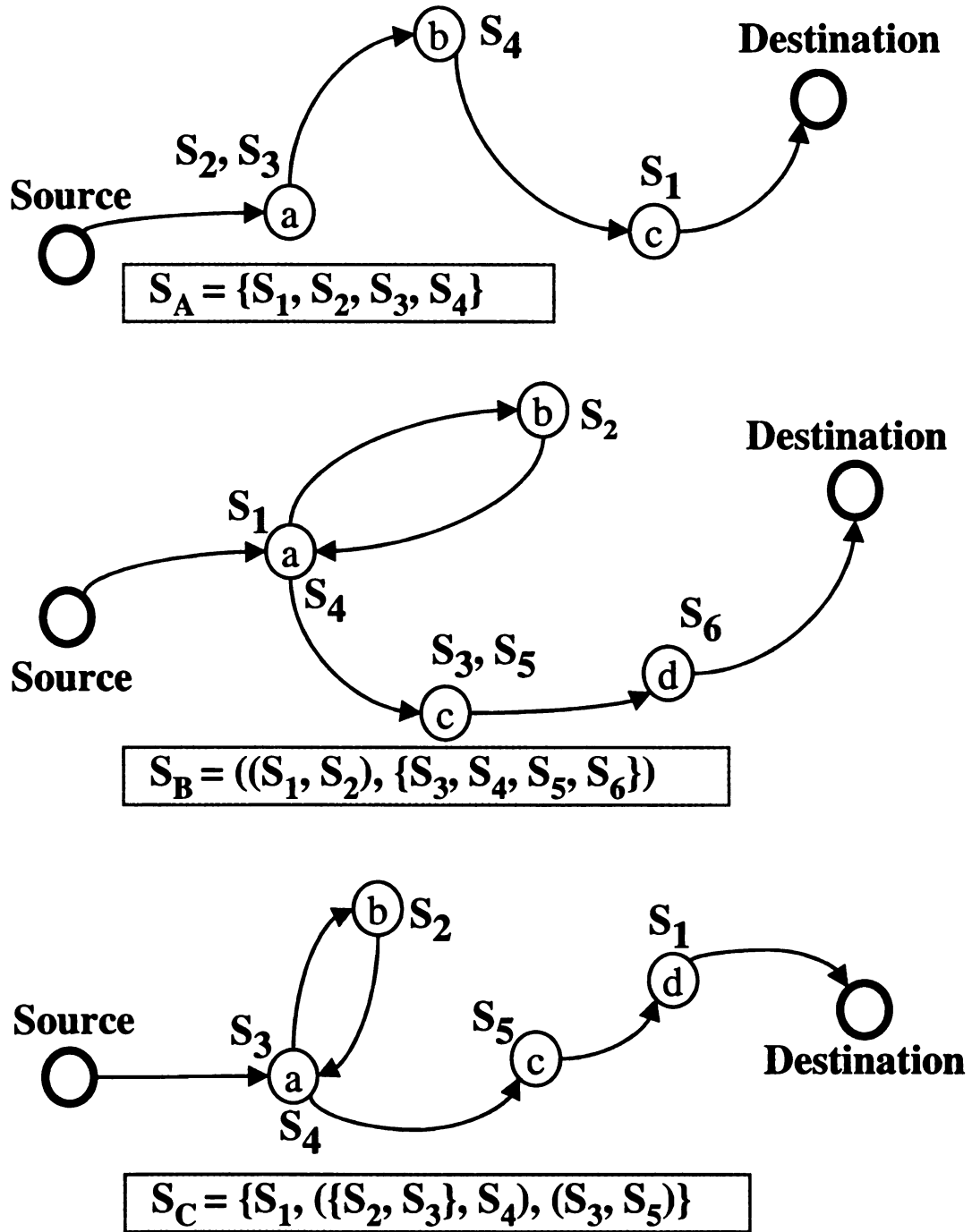


Figure 6.3: Service ordering constraints and corresponding service paths.

figures required services (possibly including existing services) and produces an overlay service graph, which implements the requested service. Composed services should consume distributed resources optimally, similarly to load-balancing and task assignment problems, while providing acceptable end-to-end quality. Further, the distributed service needs to monitor itself (alternatively, a monitoring entity could do this) and provide feedback to the service composer, when it cannot satisfactorily adapt to changing conditions.

This chapter focuses on the probing protocol and introduces a generic technique to design overlay algorithms to find high-quality service paths with minimal traffic overhead. Here, we do not intend to address the problem of adapting service graphs or sharing processed data.

Dynamis Probing Mechanism. The Dynamis algorithm is based on *distributed selection*, which applies the principle of optimality, namely, that in an optimal sequence of decisions or choices, each subsequence must also be optimal [155]. We observe that the service path composition problem satisfies the principle of optimality. That is, if we select any overlay node on an optimal service path, then the two partial service paths from that node to the endpoints must also be optimal. We use this observation to design a probing algorithm in which an overlay node drops or forwards a probe based on the quality of the partial service paths found earlier. In other words, this technique realizes distributed partial service path selection.

Figure 6.5 depicts the basic operation of the Dynamis probing mechanism, which generalizes the MCON algorithm proposed by Tang and McKinley [106] to construct multipath connections in overlay networks (Section 4.3 described the implementation of MCON in Service Clouds.) In this approach, one of the endpoints initiates probing (path-explore

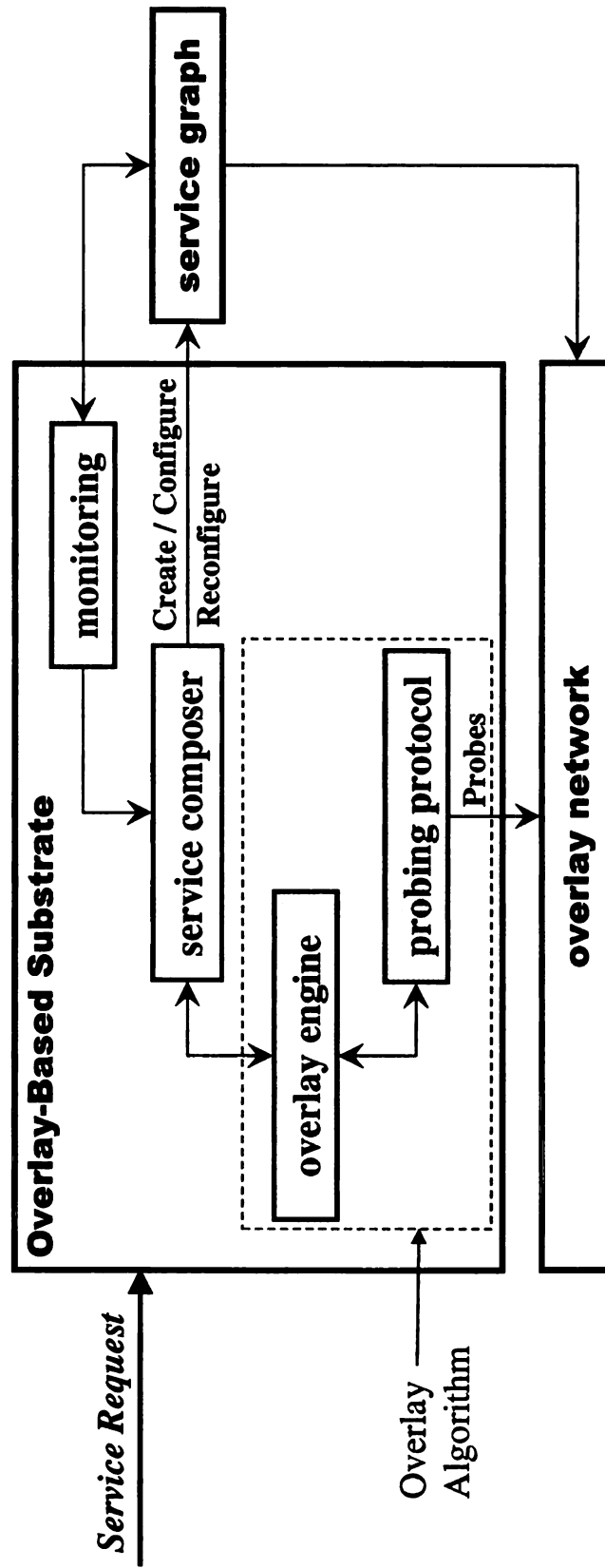


Figure 6.4: Overall service composition framework for autonomic communication.

process) by sending the probe for a service path to a subset of its neighbors in the overlay network, according to a predefined branching factor. Thereafter, probes attempt to find their way to the other endpoint. In the algorithm presented here, the destination endpoint (arbitrarily and without loss of generality) starts the path-explore process.

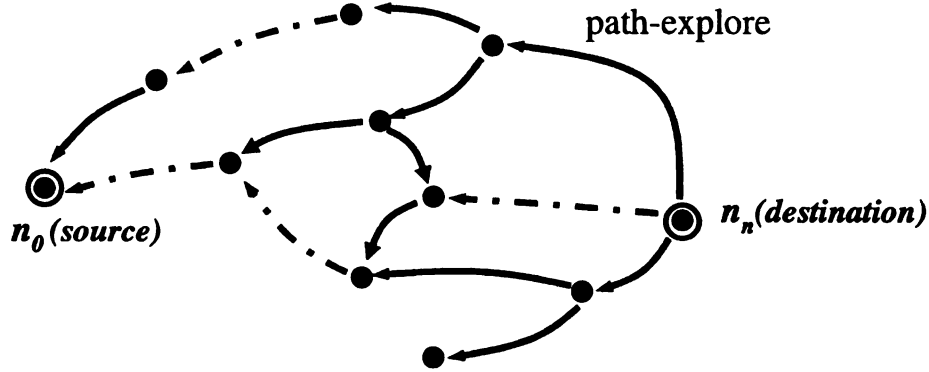


Figure 6.5: Basic operation of the composition probing algorithm.

To measure quality of a service path λ , denoted $\psi(\lambda)$, we extend the load balancing metric from Synergy [153] to include the round-trip time as a factor. Specifically,

$$\psi(\lambda) = \omega_p \sum \frac{p_{s_i}}{q_{v_i} + p_{s_i}} + \omega_b \sum \frac{b_{s_i}}{c_{v_i} + b_{s_i}} + \omega_d \frac{D}{D_{max}} n \quad (6.1)$$

where the terms of the formula are defined in Table 6.1. This metric quantifies the quality of a service path based on the processing and bandwidth load of overlay nodes on the service path, as well as its end-to-end overlay delay (half of the round-trip time). The smaller the $\psi(\lambda)$ value, the better the quality of the service path.

The key property of the Dynamis algorithm is that rather than performing selection only at an endpoint, the selection is distributed. An overlay node forwards a probe only if it describes a partial service path of significantly better quality than the quality of *compa-*

Table 6.1: Service path quality metric formula notations

Notation	Meaning
λ	service path
p_{s_i}	processing resource required for service element S_i
q_{v_i}	residual processing capacity on node v_i
b_{s_i}	uplink bandwidth required for service element S_i
c_{v_i}	residual uplink bandwidth on node v_i
D	end-to-end delay of an overlay path
D_{max}	maximum acceptable end-to-end delay
n	number of nodes in the service path
$\omega_p, \omega_b, \omega_d$	experimental cofactors

able service paths described by probes forwarded previously. As we shall demonstrate in Section 6.4, this approach, combined with a branching factor, can dramatically reduce the overhead of probing while maintaining high quality. Let us briefly describe the algorithm.

Algorithm Sketch. Upon receiving a path-explore probe, each node n_i inspects the probe and performs one of the following actions:

- (i) Node n_i drops the probe, if any of the following conditions holds:
 - (a) The service path violates the expected quality (e.g., end-to-end delay) at this point.
 - (b) Node n_i cannot satisfy resource requirements for hosting at least one of the service elements that can be added to the partial service path explored to this point (hosting).
 - (c) The quality of the partial service path explored so far is not *significantly* better (e.g., by at least 5%—as specified in the configuration) than the quality of the best *compa-*

rable partial service path described by a probe already forwarded by node n_i .

- (d) The quality of the partial service path explored so far is not better than the quality of a comparable partial service path described by a probe currently buffered to be forwarded.

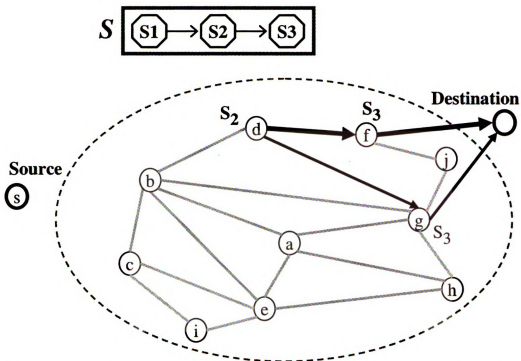
(ii) Otherwise, node n_i :

- (a) Updates the probe, adding itself to the partial service path described by the probe.
- (b) If the partial service path achieves the requested service path, node n_i announces a candidate service path. If node n_i is the target endpoint of the probe, then this announcement is local; otherwise, node n_i forwards the probe to the target endpoint.
- (c) Otherwise, node n_i buffers the probe, replacing any probe still in the buffer that describes a comparable partial service path. Node n_i periodically forwards all probes still in the buffer. The period between such forwarding operations is called an *epoch*. Node n_i forwards each probe to a subset of “qualified” neighbors. Criteria determining whether a neighbor is qualified include trust relationship, cost and availability of a particular functionality (e.g., a distributed hash table can be used to determine nodes on which a specific function is available [19]).

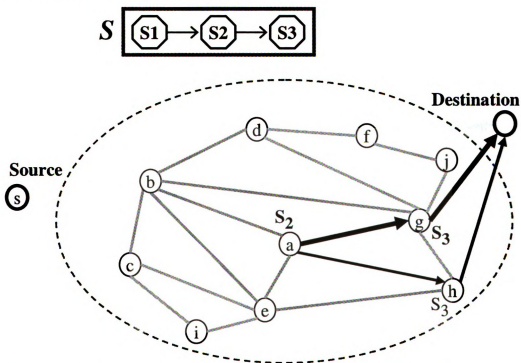
Example. Figure 6.6 demonstrates a simplified run-time operation of the Dynamis algorithm, in which the service path $S = (S_1, S_2, S_3)$ is being mapped to overlay nodes. In Figure 6.6(a) two comparable partial service paths have been found up to node d during a distributed selection epoch. The algorithm forwards only the probe describing the path

of highest quality, and then only if the quality is significantly better than the best comparable service path forwarded in a previous epoch. Let us assume that S_2 can be hosted by node d , so node d forwards $(d, f, destination)$. In a similar way, in Figure 6.6(b), $(a, g, destination)$ is selected. Then, in Figure 6.6(c) node b receives two comparable partial service paths from nodes d and a . Again, applying distributed selection (and assuming S_1 can be hosted by node b), only the partial paths with higher quality is selected if both of the probes are received in the same epoch. If the probes are received during different epochs, then the one received later is selected only if it has significantly better quality. Now, let us assume that only $(b, d, f, destination)$ is selected, which is forwarded to the source node as a candidate service path. Eventually, the source node selects the candidate with the highest quality, $(source, b, d, f, destination)$ in this example, and maps the service elements onto the corresponding overlay nodes. In approaches that do not use distributed selection, nodes typically forward all of the probes which they receive.

Parameters. Table 6.2 gives the parameters used in the algorithm. T is the buffer timeout period, or *epoch* duration. When the buffer is empty and a probe is placed in it, a timer starts that fires after T milliseconds, triggering the forwarding of all probes in the buffer. \overline{B} , \overline{H} , and \overline{W} are branching factor parameters that control the overhead of the probing in three respective ways: budgeted, limited-hop, and bounded. In *budgeted* forwarding, each node forwards a probe to \overline{B} qualified neighbors selected at random. In limited-hop forwarding, a probe is discarded if it has not found a service path after traversing \overline{H} overlay nodes. In *bounded* forwarding, each node forwards at most \overline{W} probes for a service composition session. Finally, \underline{Q} specifies the minimum service path quality improvement expected, relative to the quality of a comparable service path in a probe forwarded previously, in

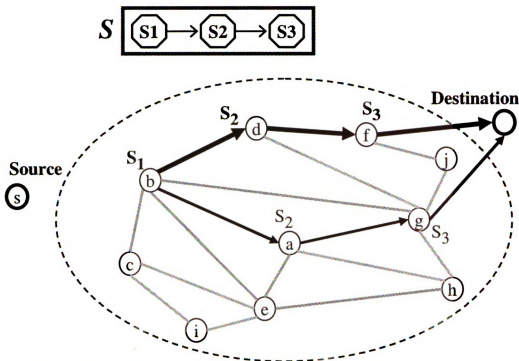


(a) node d selecting the partial service path passing through node f , which has higher quality than the comparable one passing through node g

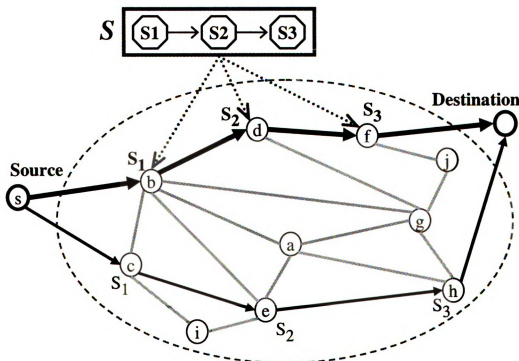


(b) node a selecting the partial service path passing through node g , which has higher quality than the comparable one passing through node h

Figure 6.6: Example run of the Dynamis probing algorithm: mapping $S = (S_1, S_2, S_3)$ to the overlay network (continued on the next page).



(c) node b selecting the partial service path passing through nodes d and f , which has higher quality than the comparable one passing through nodes a and g



(d) the source node s selecting $(source, b, d, f, destination)$, the candidate service path with the highest quality

Figure 6.6: Example run of the Dynamis probing algorithm: mapping $S = (S_1, S_2, S_3)$ to the overlay network.

Table 6.2: Dynamis algorithm parameters.

Notation	Meaning
T	buffer time-out (epoch duration)
\overline{H} upper bound	number of hops a probe can traverse
\overline{W} upper bound	number of probes forwarded at each node
\overline{B} upper bound	number of forwards for a probe at each node
Q	quality improvement expected in order to forward a probe

order to forward a probe in question.

6.3 Dynamis Algorithm

In this section, we describe the operation of the Dynamis algorithm pseudocode presented in Figure 6.7. The algorithm consists of two main parts: probe processing and probe buffering. The probe processing procedure determines whether to drop a probe or to forward it. It drops a probe if the node can host at least one service element from the service path. Otherwise, it augments the information of the current node to the probe and schedules the probe to be forwarded. If the probe does not complete a service path, it is buffered; otherwise, the candidate service path is announced immediately. The probe buffering procedure periodically checks the buffer of probes and either forwards the buffered probes or drops them, based on the specified branching factors and the current statistics of the probe forwarding on the node. The details of the Dynamis algorithmic procedure follow.

In the probe processing, the receiving node checks whether it can host at least one of the service elements that are to be executed next (line 2). Note that the source and destination nodes may also execute a service element; thus, the path-explore initiating node can also

Algorithm: Dynamis

Probe Processing

```
1 ) on receiving probe probe on node n:
2 )   if n cannot host probe.servicePath then
3 )     drop probe and return;
4 )   if ( probe.quality < expectedQuality(probe) or
         probe.quality < buffer.bestQuality(probe) ) then
5 )     drop probe and return;
6 )   probe := annex(n, probe);
7 )   if probe.serviceGraph is achieved then
8 )     if ( probe.target == n ) then announce probe locally
9 )     else send probe to probe.target
10 )  else
11 )    assign probe to buffer
12 )  return;
```

Probe Buffering

```
13) on probe assignment to buffer on node n:
14)   if buffer is empty then start buffer.timer
15)   buffer.setBufferBestQuality(probe);
16)   return;

17) on buffer.timer timeout on node n:
18)   for each probe probe in the buffer
19)     if ( numberOfHopsTraversed(probe) >  $\overline{H}$  or
         n.numberOfWorkForwards(probe.sessionID) >  $\overline{W}$  ) then
20)       remove probe from buffer and return;
21)     Forward probe to not more than  $\overline{B}$  qualified nodes in n.neighbors
22)     Increment n.numberOfWorkForwards(probe.sessionID)
23)     setExpectedQuality(probe, probe.quality * (1 +  $\underline{Q}$ ))
24)     remove probe from buffer
25)   end for
26)   return;
```

Figure 6.7: Dynamis algorithm pseudocode.

process the probe, although, we have not considered this in the implementation—rather, the destination node simply forwards the probe describing the service path to its neighbors. If the node cannot be a host for the service graph, the probe is dropped (line 3).

If the quality of the (partial) service path is not significantly better, that is, at least \underline{Q} percent improvement, than the best *comparable* partial service path already forwarded at this node, or it is not better than the quality of a comparable service path traversed by a probe currently buffered to be forwarded, the probe is dropped (lines 4-5). Otherwise, the current node is added to the probe and the information carried in the, regarding the partial service path, is updated (line 6). Afterwards, the probe is scheduled to be forwarded (lines 7-11). If the current node is added to the service path and this completes a candidate service path (line 7), then the candidate service path is announced (lines 8-9). If current node is the source, then this announcement is local; otherwise, the updated probe is sent to the source directly. If the traversed service path is still partial after addition of current node, the probe is assigned to the buffer.

The probe buffering operates as follows (lines 13-26):

1. Upon assignment of a probe to the buffer, the probe replaces any comparable probe in the buffer. If the buffer is empty, a timer starts that expires after T milliseconds.

The assignment also keeps track of the quality of buffered probes (lines 13-16).

2. Upon the expiration of the timer, the probes in the buffer are handled as below.

- (a) If the specified branching factors do not support further forwarding of this probe, the probe is dropped (lines 19-20).

- (b) A probe is forwarded to a number of qualified neighbors based on the branching factor on the maximum number of forwards allowed for a probe at each node (line 21). Accordingly, the node keeps information about the forwarded probe (line 22-24). This procedure includes setting the minimum quality expected of comparable service paths traversed by probes which will be received in next epochs in order to be considered for forwarding by the probe processing procedure (line 23).

6.4 Experimental Evaluation

In this section, we assess the performance of the proposed method to compose services on PlanetLab [95]. The ultimate goal is to compose as many as possible high-quality service paths, while having low probing overhead. We have used the Service Clouds prototype to apply Dynamis to different probing strategies, using the framework and extending existing components in the prototype toolkit. In these experiments we assume all service elements need to be executed in order (ordered service path).

6.4.1 Test Setup and Procedure

We consider establishment of service paths between two nodes on the Internet. We first show that the proposed approach significantly reduces probing overhead by incorporating distributed selection (rather than selection at an endpoint), while still finding high-quality service paths with small end-to-end delay. We use the Formula 6.1 (Section 6.2) to measure the quality of a service path. Next, we evaluate the quality of the selected service paths

in different approaches and various configurations in terms of end-to-end delay and load distribution. In particular, we assess performance under different system loads, and the effect of Q (the minimum quality improvement expected to forward a probe) and T (the buffer timeout that specifies the duration of an epoch) parameters in distributed selection.

We measure the quality of the best service path found and the corresponding probing overhead. The test procedure cycles through source-destination pairs of nodes, submitting service requests to the system and allocating resources. This implementation realizes a simplified version of the Dynamis algorithm, which assumes two elements of the same service path do not execute on the same overlay node (hence, probes are not forwarded to nodes which they have already traversed). We have implemented a simple protocol to reserve virtual resources on the nodes, because, unlike Grids, resource management in overlay networks is often not supported or is not sufficiently fine-grained. For example, the Sirius scheduling service on PlanetLab allows users to reserve resources only at a coarse-grained level (e.g., 25% of CPU capacity). It does not support finer allocation of the reservation among user applications. The virtual resource plane proposed here represents the status and the acquisition of the resources in a normalized scale. Moreover, it can facilitate finer resource management when resource consumptions of service elements are characterized in advance (e.g., by profiling them off-line). If fine-grained resource management is available on an underlying distributed platform, the virtual plane can interact with the particular resource reservation framework transparently to the upper layers of the overlay infrastructure (cross-layer cooperation). The virtual resource plane allows to take into account the resource reservation behavior of different algorithms as composed services load the overlay network.

Table 6.3 shows the setting of parameters in the experiments. In case of multiple values for a parameter, the one in the parentheses is the default; unless explicitly stated, the default value has been used. In these tests, until an ongoing service composition procedure is complete, another service composition request is not issued. All service path specifications contain four ordered service elements, which consume the same amount of resources: 20 units of CPU processing power and 200Kbps of bandwidth. Finally, Table 6.4 contains the list of 28 nodes used in the experiments. The capacity of each node is 100 units CPU processing power and 1024Kbps uplink bandwidth. The results presented are the average of five samples of service composition between each pair of nodes.

Table 6.3: Configuration of parameters in the Dynamis experiment.

Parameter	Value
distributed service selection	(enabled) , disabled
initial resource loads (CPU and bandwidth)	from (0%) to 60%
T (buffer timeout)	(500) , 1500 , 2000 msec
\bar{H} (total number of hops a probe can traverse)	N/A
\bar{W} (total number of probes forwarded at each node)	(unlimited), 3000
\bar{B} (total number of forwards for a probe at each node)	unlimited
Q (quality improvement expected to forward a probe)	(0%) , 2% , 3% , 5% , 10%
p_{s_i} (processing requirement for each service element)	20 units of normalized CPU time
b_{s_j} (bandwidth requirement for each service element)	200 Kbps
D_{max} (acceptable end-to-end delay)	300 msec
$\omega_p, \omega_b, \omega_d$ cofactors	1.0

6.4.2 Performance Analysis

These tests compare four major strategies:

- *populated - nopt*: composes a service path using nodes that already host the required services; does not use distributed selection.

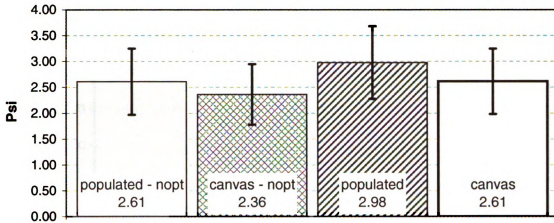
Table 6.4: List of nodes in the Dynamis experiment.

ID	Node Name	Location	Function	Pair #
0	planet1.scs.cs.nyu.edu	USA – NY	F3	1
1	planetlab01.cs.washington.edu	USA – WA	F2	
2	planetlab1.iii.u-tokyo.ac.jp	Japan	F1	2
3	planetlab1.ls.fi.upm.es	Spain	F3	
4	planetlab2.cs.uoregon.edu	USA – OR	F3	3
5	planetslug1.cse.ucsc.edu	USA – CA	F2	
6	planetlab1.cs.ubc.ca	Canada	F1	4
7	planetlab-01.bu.edu	USA – MA	F3	
8	planetlab-1.cs.colostate.edu	USA – CO	F4	5
9	planetlab1.cs.duke.edu	USA – NC	F1	
10	planetlab1.diku.dk	Denmark	F1	6
11	planetlab-1a.ics.uci.edu	USA – CA	F2	
12	planetlab1.csail.mit.edu	USA – MA	F2	7
13	planetlab1.hiit.fi	Finland	F3	
14	planetlab1.tamu.edu	USA – TX	F2	8
15	kc-sce-plab1.umkc.edu	USA – MO	F1	
16	planetlab1.cs.wisc.edu	USA – WI	F4	9
17	planetlab1.cslab.ece.ntua.gr	Greece	F1	
18	planetlab1.uta.edu	USA – TX	F4	10
19	planetlab1.cs.purdue.edu	USA – IN	F2	
20	planetlab1.lsd.ufcg.edu.br	Brazil	F1	
21	planet2.cs.huji.ac.il	Israel	F3	
22	planetlab1.eecs.umich.edu	USA – MI	F4	
23	planetlab1.cs.uit.no	Norway	F4	
24	planetlab1.cs.unc.edu	USA – NC	F4	
25	planetlab1.xeno.cl.cam.ac.uk	UK	F4	
26	pl1.ucs.indiana.edu	USA – IN	F3	
27	planetlab2.sics.se	Sweden	F2	

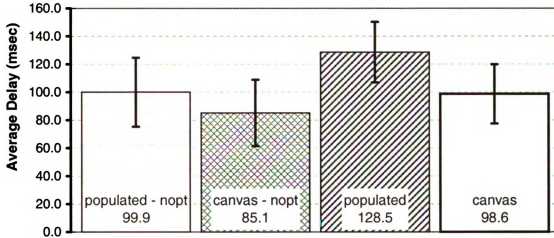
- *canvas - nopt*: considers the overlay as a blank computational canvas (supported in the Service Clouds model), so any service can be mapped to any node with sufficient resources; does not use distributed selection.
- *populated*: composes a service path using nodes that already host required services; uses distributed selection.
- *canvas*: considers the overlay as a blank computational canvas and uses distributed selection.

Baseline evaluation. Figures 6.8(a) and 6.8(b) show the average quality and end-to-end delay for each of the four strategies (95% confidence intervals are included). While the values for the distributed selection cases (referred to henceforth as the *populated strategy* and the *canvas strategy*) are slightly higher (therefore worse) than those for the first two cases, the differences are not significant. On the other hand, Figures 6.8(c) and 6.8(d) show that the probing overhead is dramatically reduced by using distributed selection (by over 93% in the populated strategy, and over 97% in the canvas strategy).² Fig. 6.8(a) and Fig. 6.8(b) also show that with distributed selection, the canvas strategy can improve the quality of service composition with a reasonable increase in probing overhead. We note that the canvas - nopt strategy (which does not use distributed selection) generated such a large number of probes that many nodes ran out of memory—to avoid this situation we set the maximum number of probes forwarded by each node in a service composition session for a pair of source-destination \overline{W} , to 3000 in the canvas - nopt strategy.

² In the populated - nopt and populated strategies, we are not considering the overhead of looking up location of services in the overlay. Thus, the probing overhead is likely higher in those strategies, depending on the use of service look-up methods such as DHT.



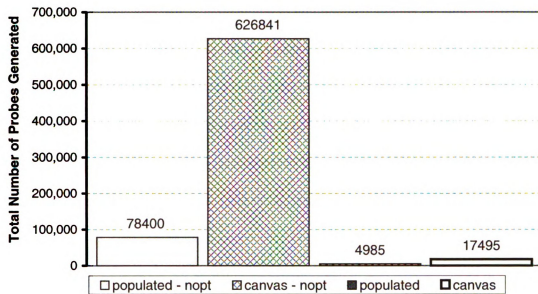
(a) average quality of selected service paths



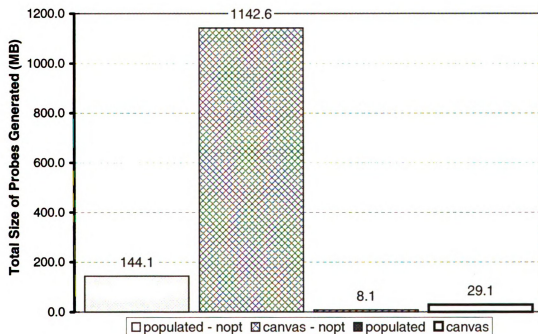
(b) average end-to-end delay of selected service paths

Figure 6.8: The quality and the end-to-end delay of selected service paths vs. overhead of probing in the four different strategies (continued on the next page).

Effect of T and Q . Figure 6.9 shows the effect of the parameter T on the performance in terms of the end-to-end delay and probing overhead in canvas strategy. These plots show no significant change as T is set to the different values of 500, 1500, and 2000 msec. This observation is expected, since each node compares the quality of a partial service path both to all other ones received during the same epoch, and to those forwarded in previous epochs. Thus, unless the T is so small that not many probes are received during an epoch, increasing



(c) total number of generated probes



(d) total size of generated probes

Figure 6.8: The quality and the end-to-end delay of selected service paths vs. overhead of probing in the four different strategies.

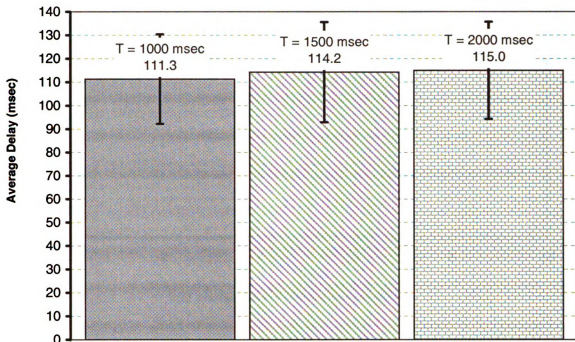
the buffer timeout will not change the behavior of the strategies. The experiments show that the value of 500 *msec* for T is sufficient to significantly reduce the probing overhead.

Figure 6.10 shows the effect of the parameter \underline{Q} on the end-to-end delay and probing overhead. We expect increasing \underline{Q} to reduce the probing overhead as well as the quality of the selected service path at the endpoint, since a node drops a probe if the probe describes partial service path with a quality improvement lower than \underline{Q} . However, the results show that increasing \underline{Q} (up to 10%) has little effect on overhead, but does increase end-to-end delay.

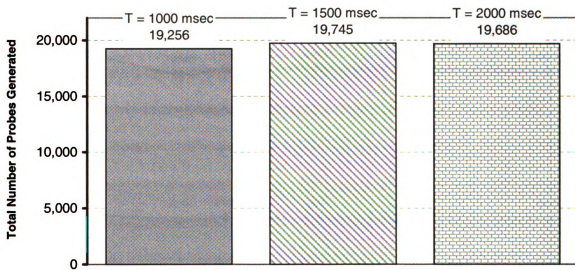
We can conclude from the above observations that an epoch duration of 500 *msec* is sufficient to significantly reduce the probing overhead of composing high-quality service paths, without the need for a positive \underline{Q} at least within the realm of these experiments.

Resource consumption. To evaluate the performance of the canvas strategy versus the populated strategy under different system loads, we impose an initial load on the CPU and bandwidth of all nodes when starting a test. Figure 6.11 shows the mean quality and end-to-end delay of the service paths found. The canvas strategy produces service paths of significantly better quality than the populated strategy, especially in terms of the end-to-end delay as the load reaches 60%. Running *t*-test on the end-to-end delays at 60% CPU load indicates a *p*-value of 0.012, which means this difference is significant.

The above observation shows the overall effect of the load on the performance of the two strategies. However, since it considers the average performance over 10 different pairs of nodes, we need to verify this evaluation consistently holds for a significant number of pairs (rather than the performance in one pair dominating the average). Accordingly, we assess the average performance improvement for each pair. Figure 6.12 plots the average

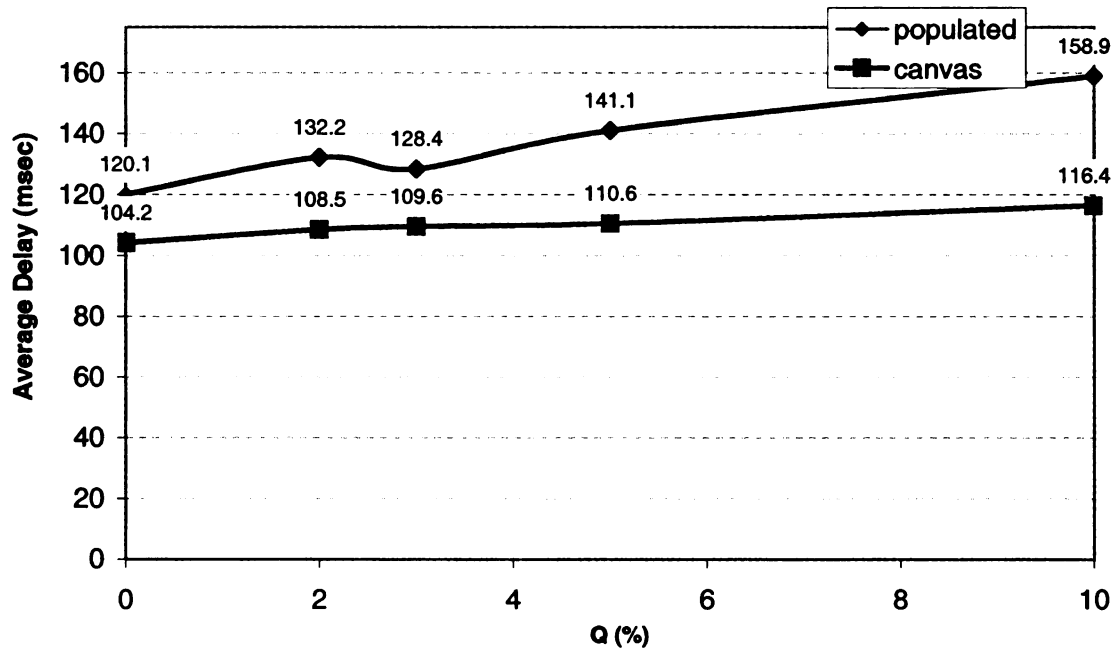


(a) average delay of selected service paths

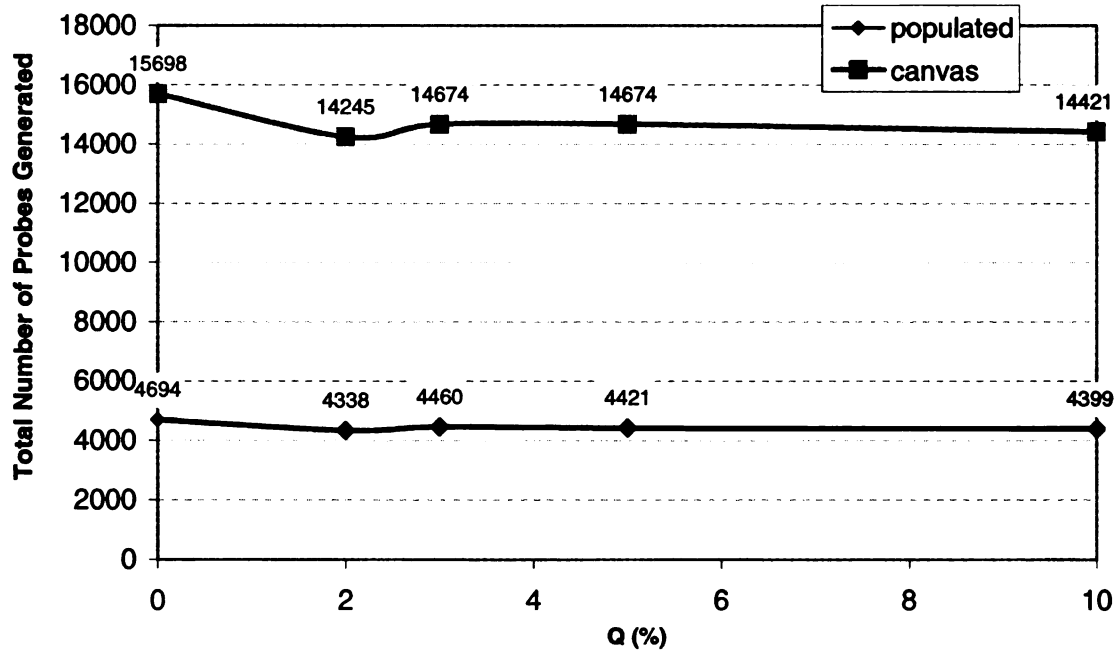


(b) total number of generated probes

Figure 6.9: Effect of T on the service path delay and probing overhead (canvas strategy with $Q = 5\%$).



(a) average delay of selected service paths



(b) total number of generated probes

Figure 6.10: Effect of Q on the service path delay and probing overhead.

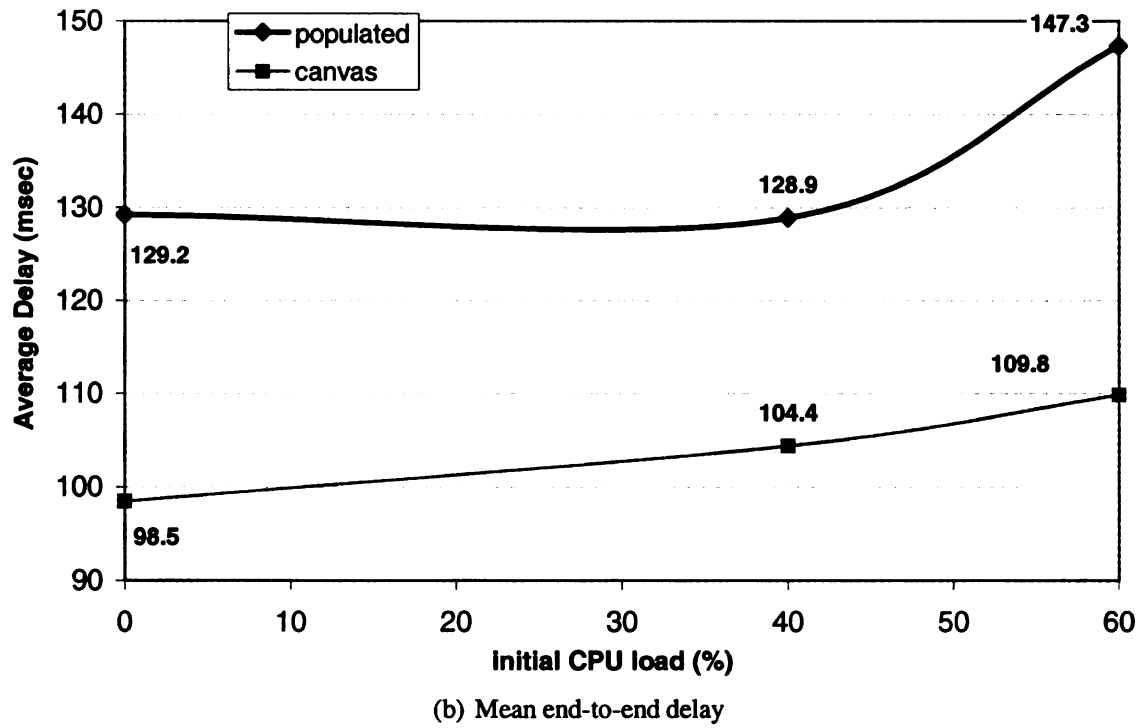
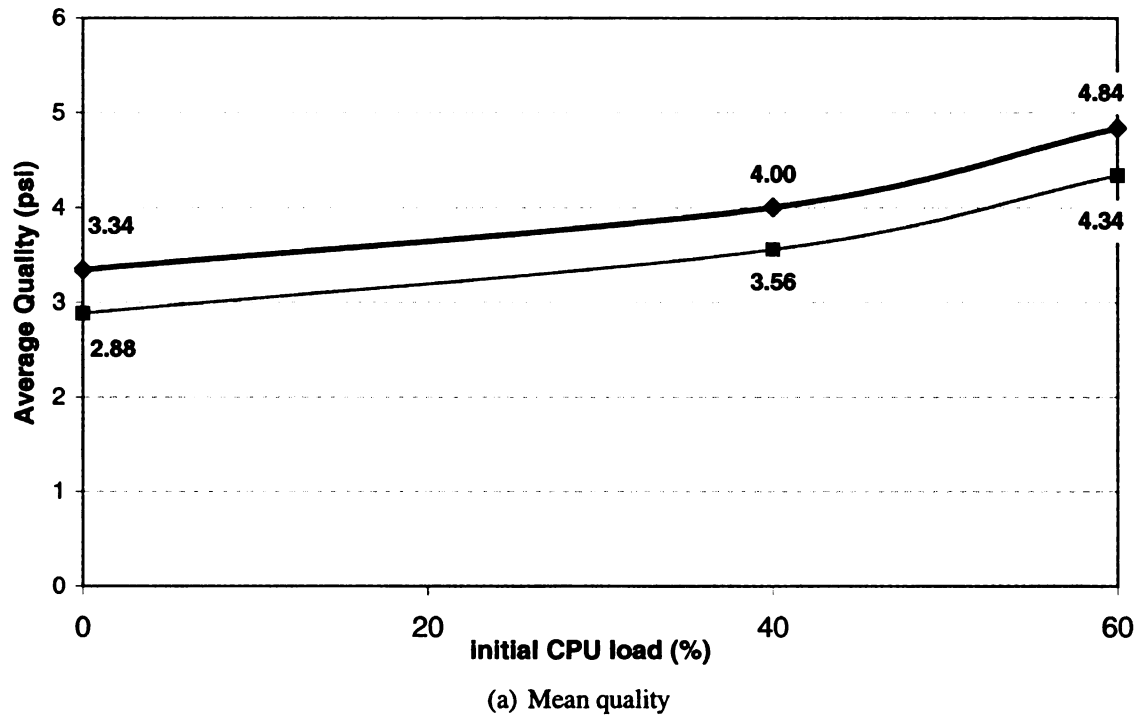


Figure 6.11: Quality and end-to-end delay of service paths as load increases.

performance improvement for each pair with a confidence interval of 90%. Excluding pair #1, since one of the nodes became unavailable in the middle of the second sampling, this is the average of four or five samples for each pair (except for pair #10 in populated mode at 60% load, in which each point on the plot is an average of only two samples).

We observe that at 60% CPU load, in 8 out of 10 pairs the canvas strategy outperforms the populated strategy. However, in only 50% of these cases the confidence intervals of the two strategies are not overlapping, that is, the difference is significant. On the other hand, since the results were collected over several days, fluctuation in network conditions can produce high variance even in samples belonging to the same pair. To assess the performance of the strategies over a shorter period of time, we investigate the performance for each pair of nodes during the same arbitrary sampling window. Figure 6.13 shows plots for each pair in the third sampling window (as mentioned above, one of the nodes was in failure mode from the middle of the second sampling window, thus no data is available for pair #1). We observe that at 60% CPU load, in 8 out of 9 cases, the canvas strategy outperforms the populated strategy.

In summary, at 60% CPU load, the canvas strategy significantly outperforms the populated strategy in 80% of cases. In the remaining 20% of cases, the two strategies have no significant difference in performance (we see that in Figure 6.12 the populated strategy never yields significantly better performance). As Figure 6.11 shows, overall the canvas strategy improves the quality at least 10%, with over 25% improvement in end-to-end delay, at 60% CPU load.

The better performance of the canvas strategy under heavy loads is due to the behavior of resource consumption. Specifically, the canvas strategy results in load distribution

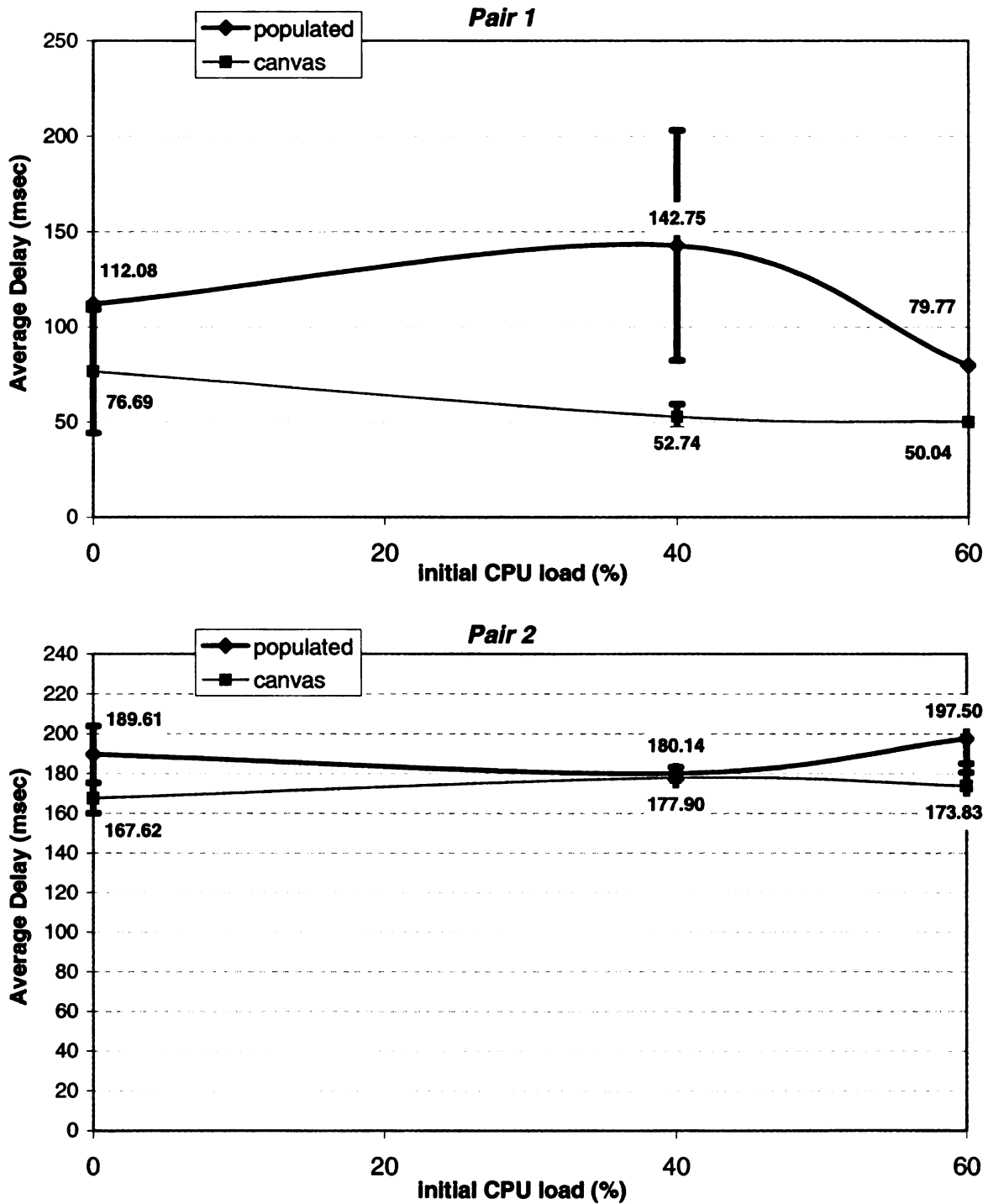


Figure 6.12: Average end-to-end delay of service paths as load increases: pairs 1 and 2 (continued on the next page).

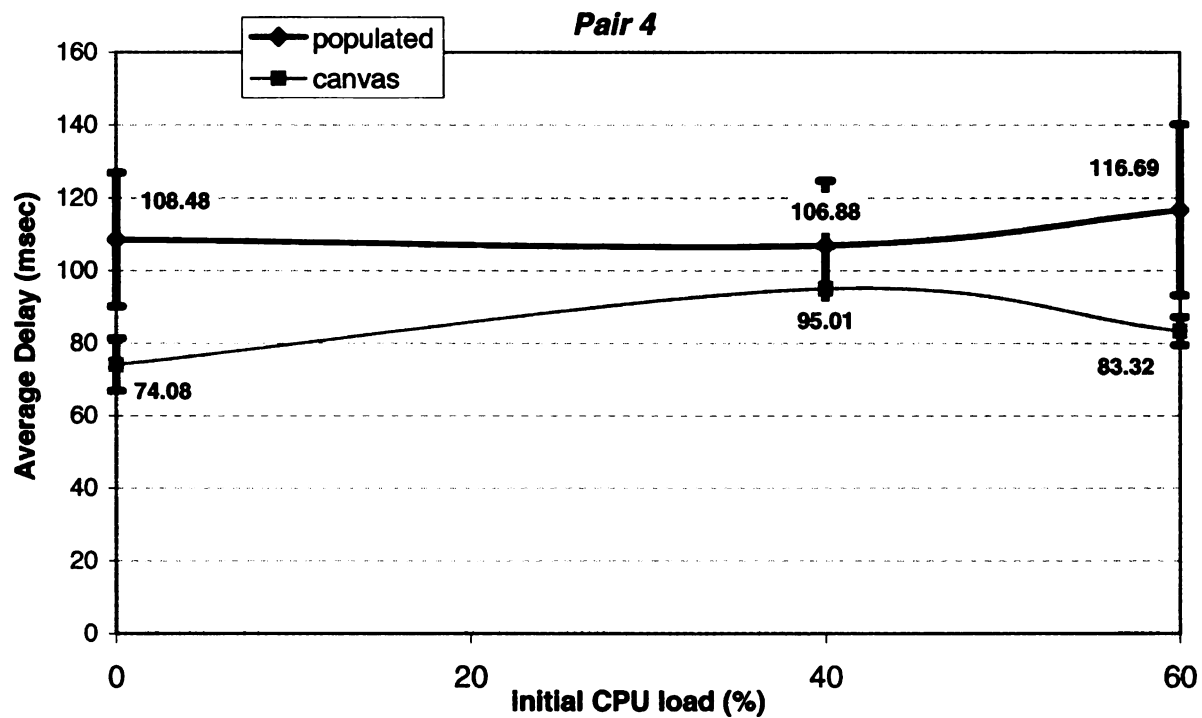
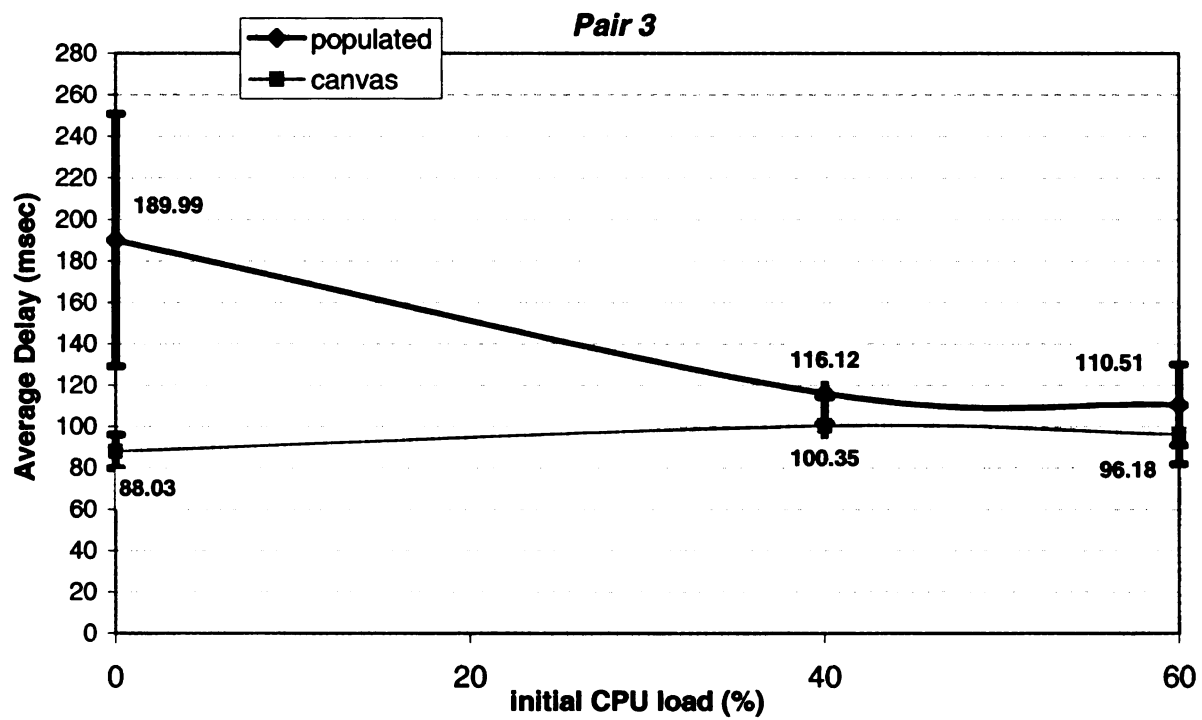


Figure 6.12: Average end-to-end delay of service paths as load increases: pairs 3 and 4 (continued on the next page).

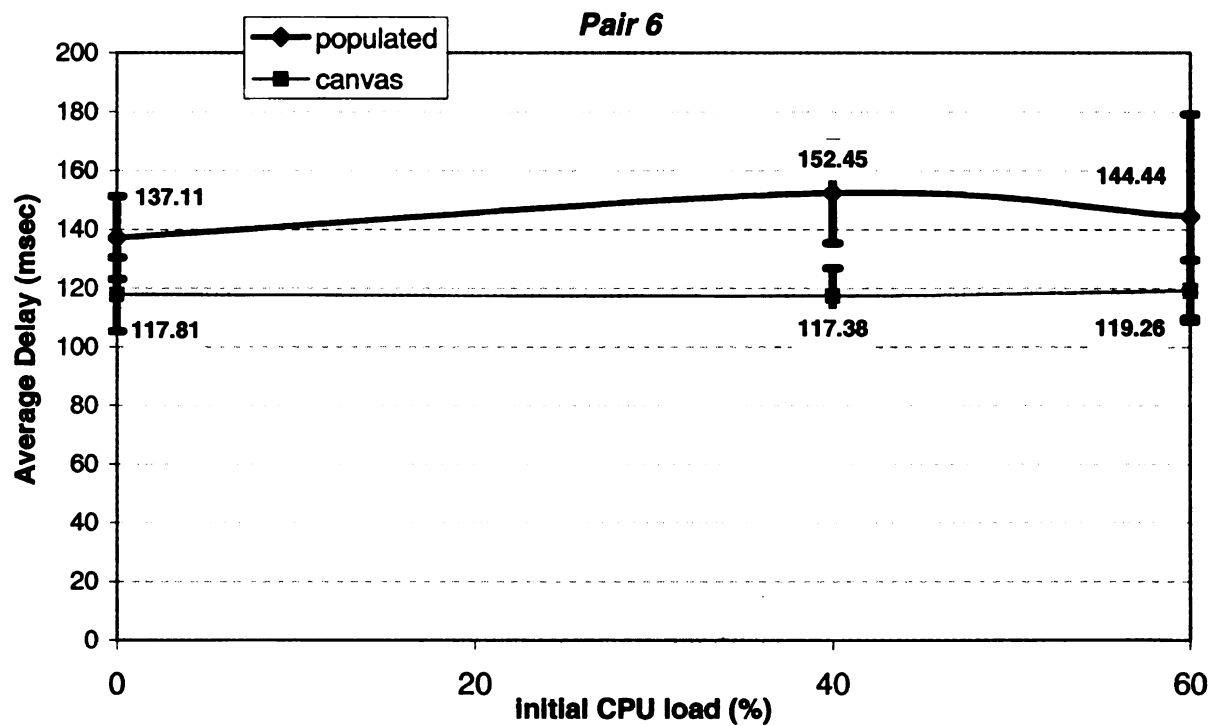
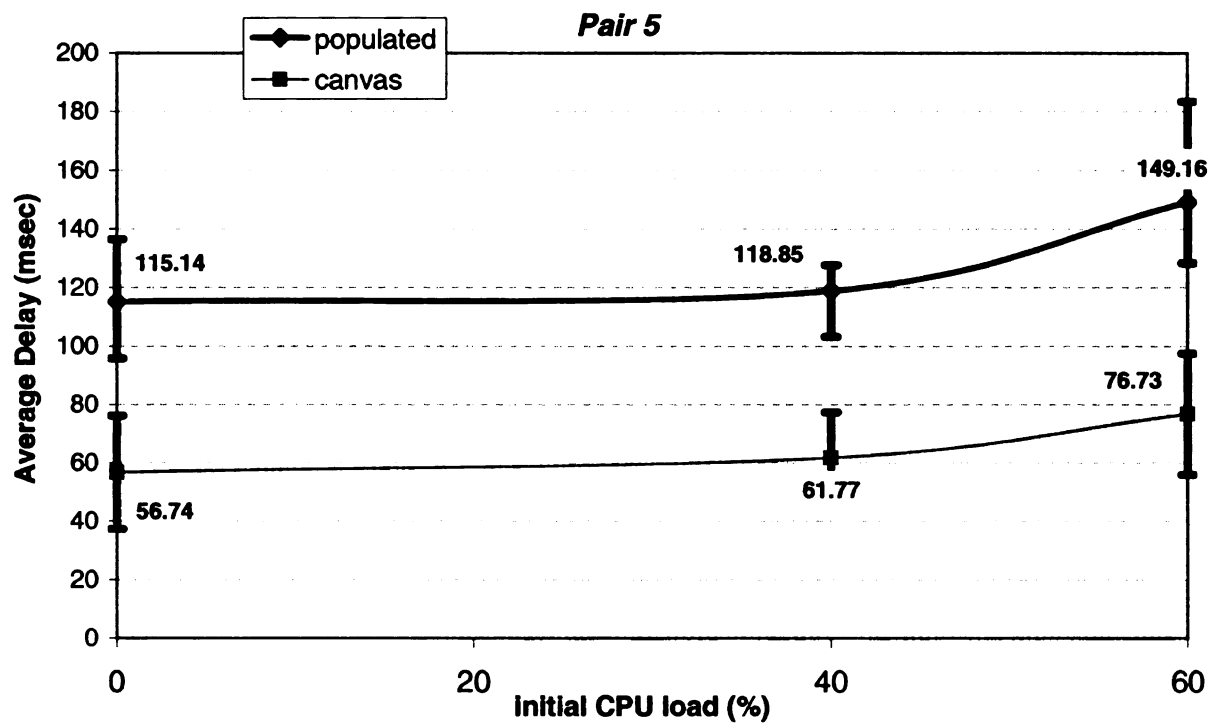


Figure 6.12: Average end-to-end delay of service paths as load increases: pairs 5 and 6 (continued on the next page).

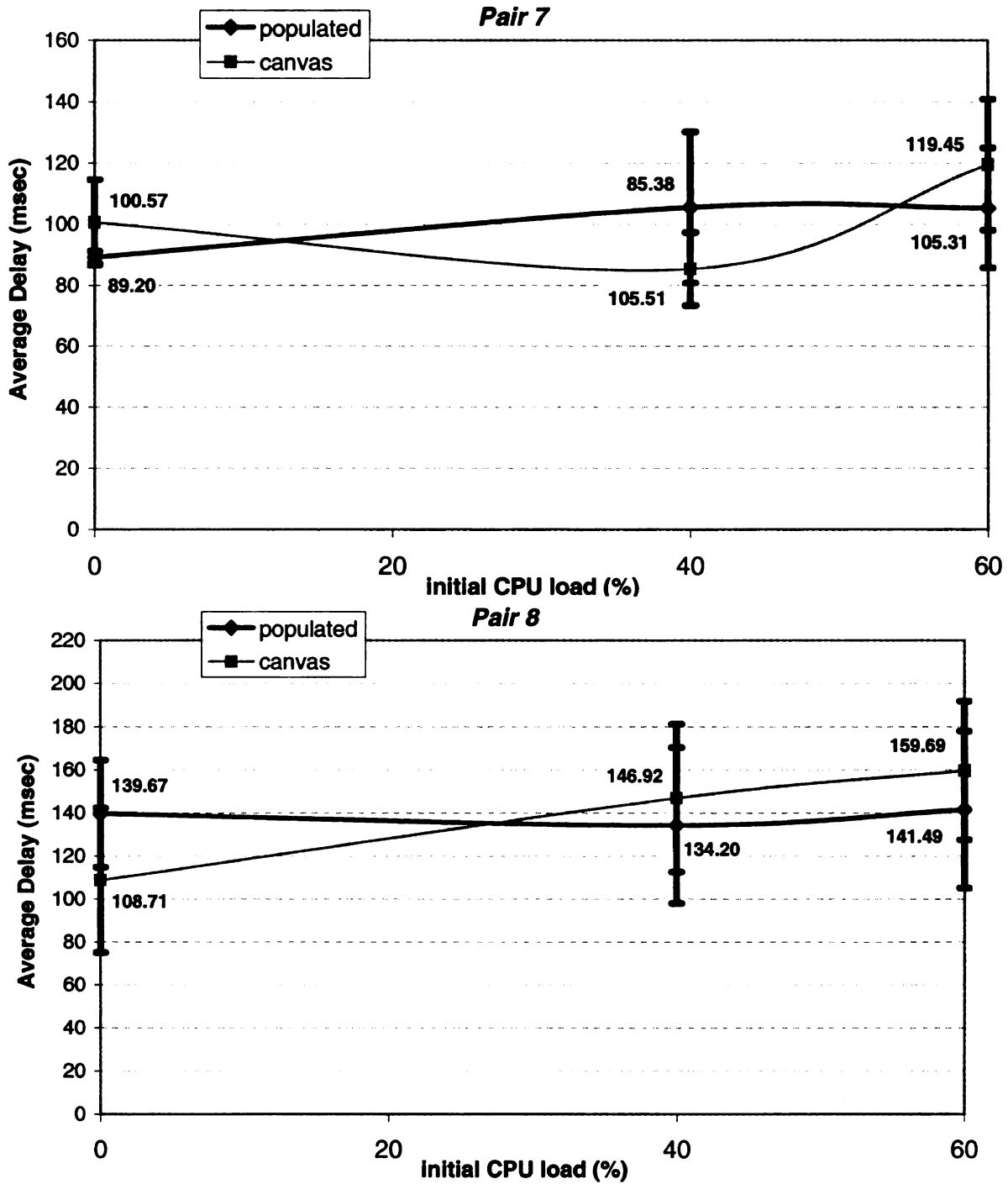


Figure 6.12: Average end-to-end delay of service paths as load increases: pairs 7 and 8 (continued on the next page).

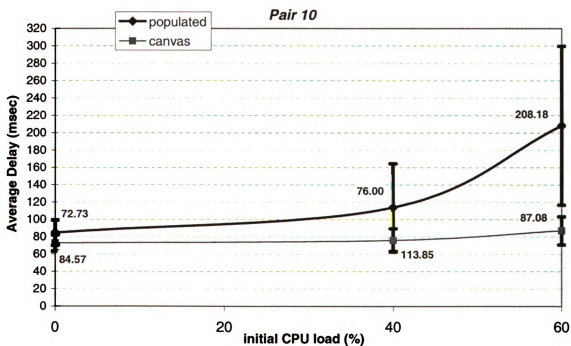
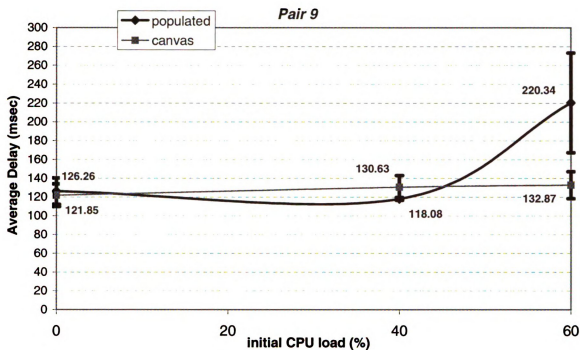


Figure 6.12: Average end-to-end delay of service paths as load increases: pairs 9 and 10.

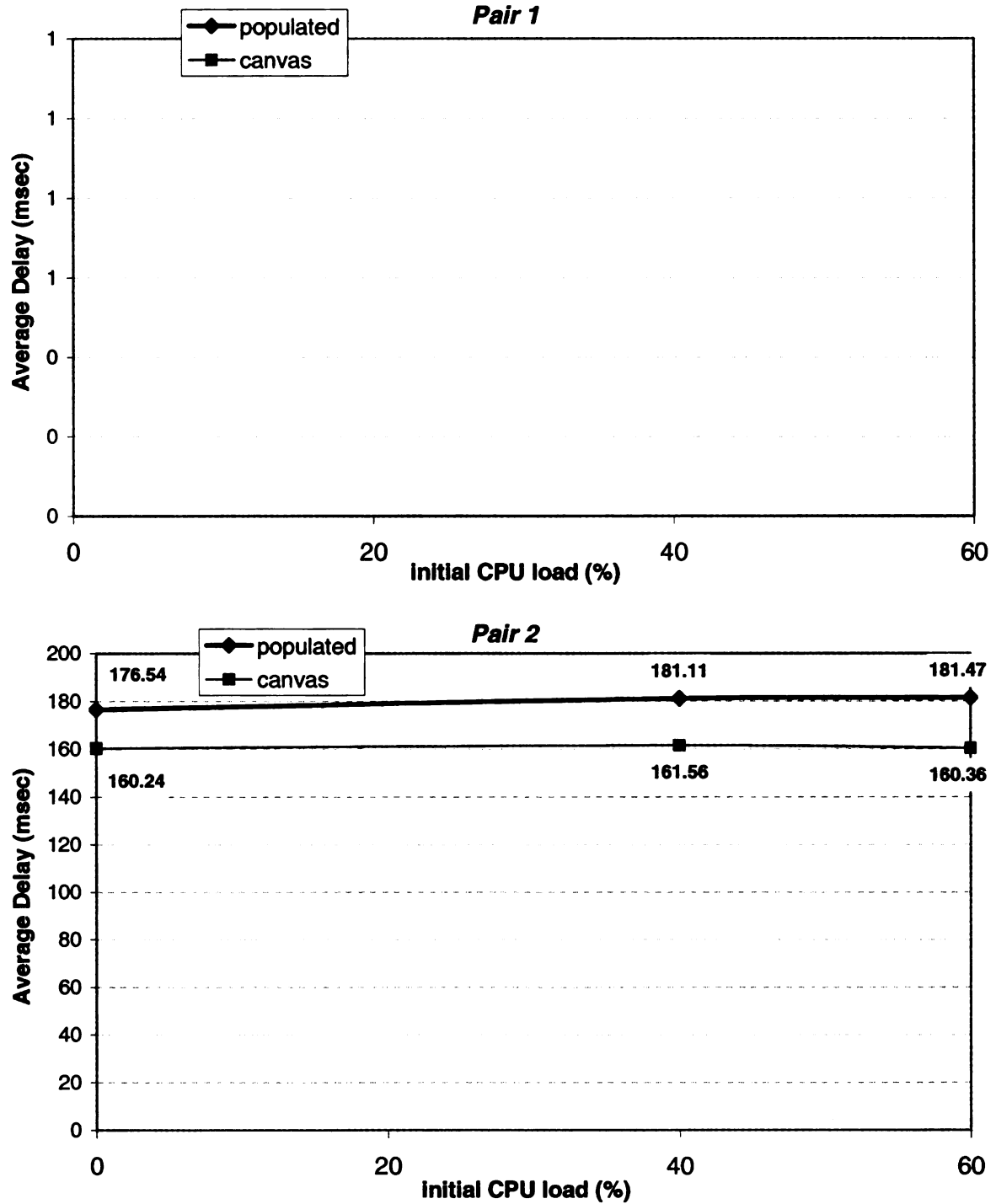


Figure 6.13: A sample of end-to-end service path delay as load increases: pairs 1 and 2 (continued on the next page).

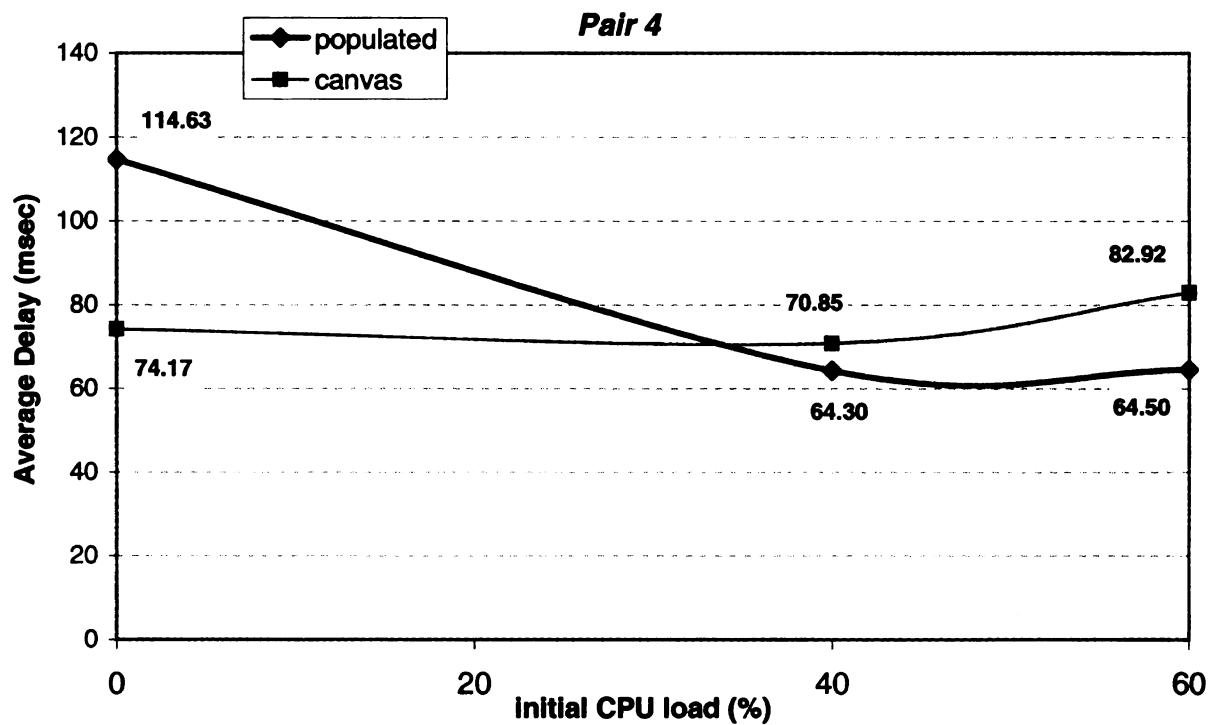
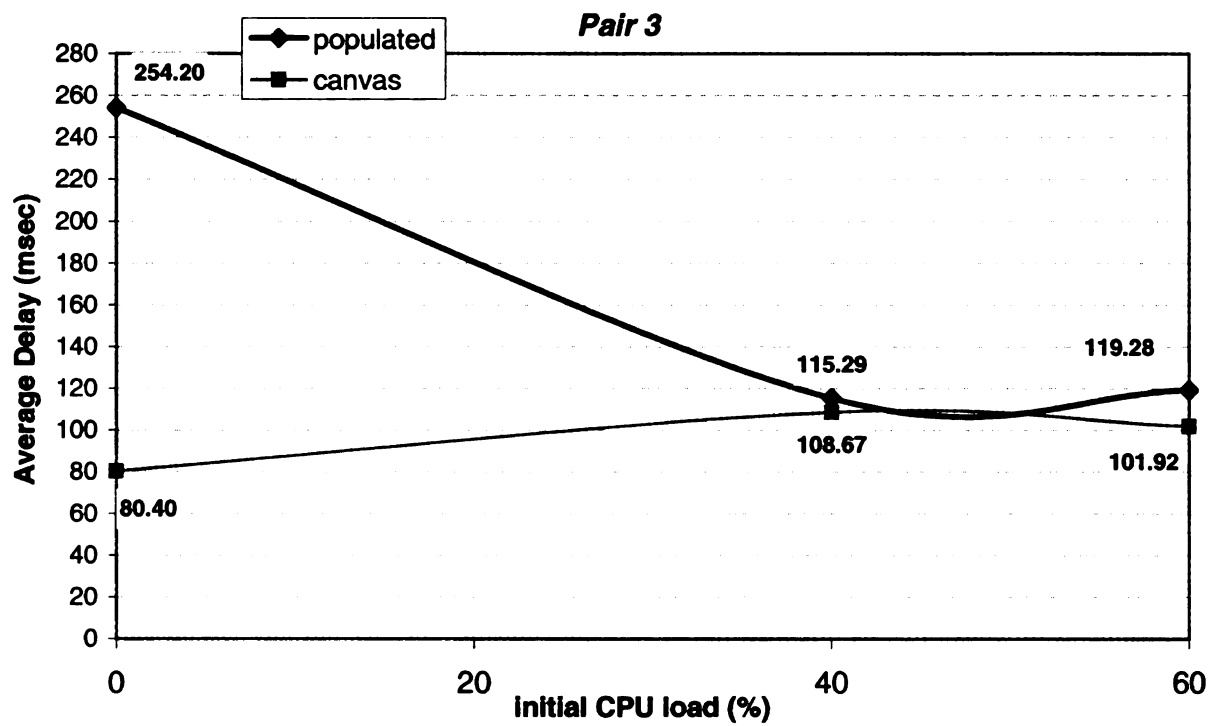


Figure 6.13: A sample of end-to-end service path delay as load increases: pairs 3 and 4 (continued on the next page).

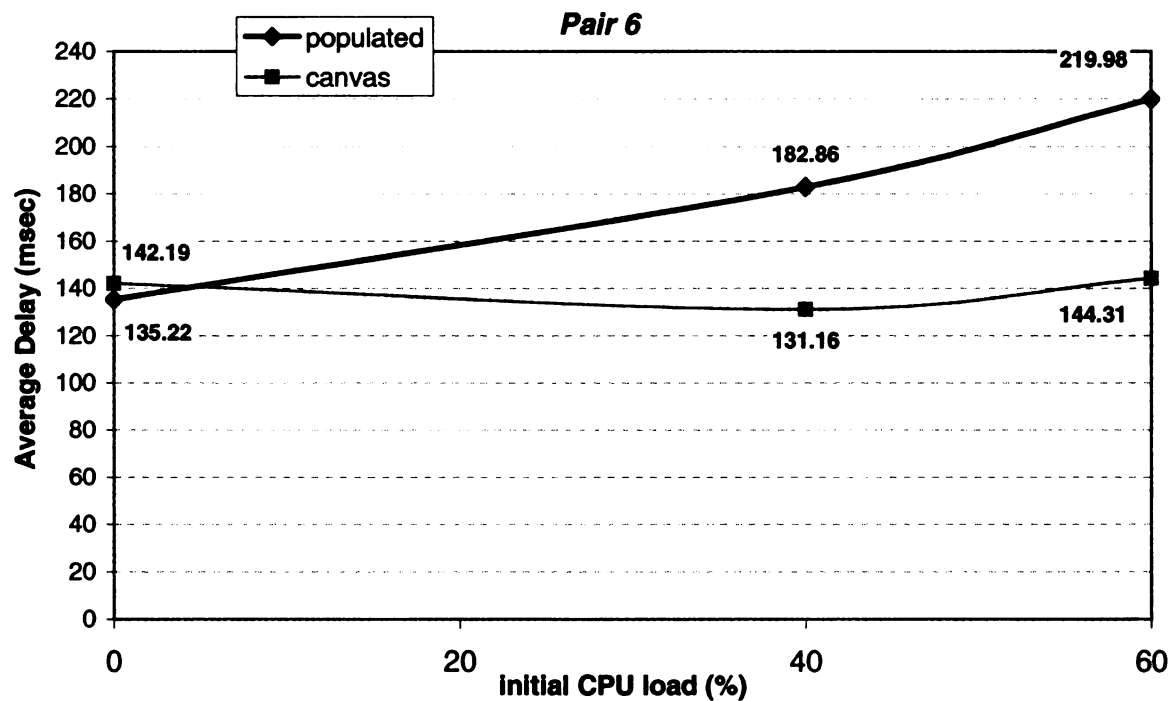
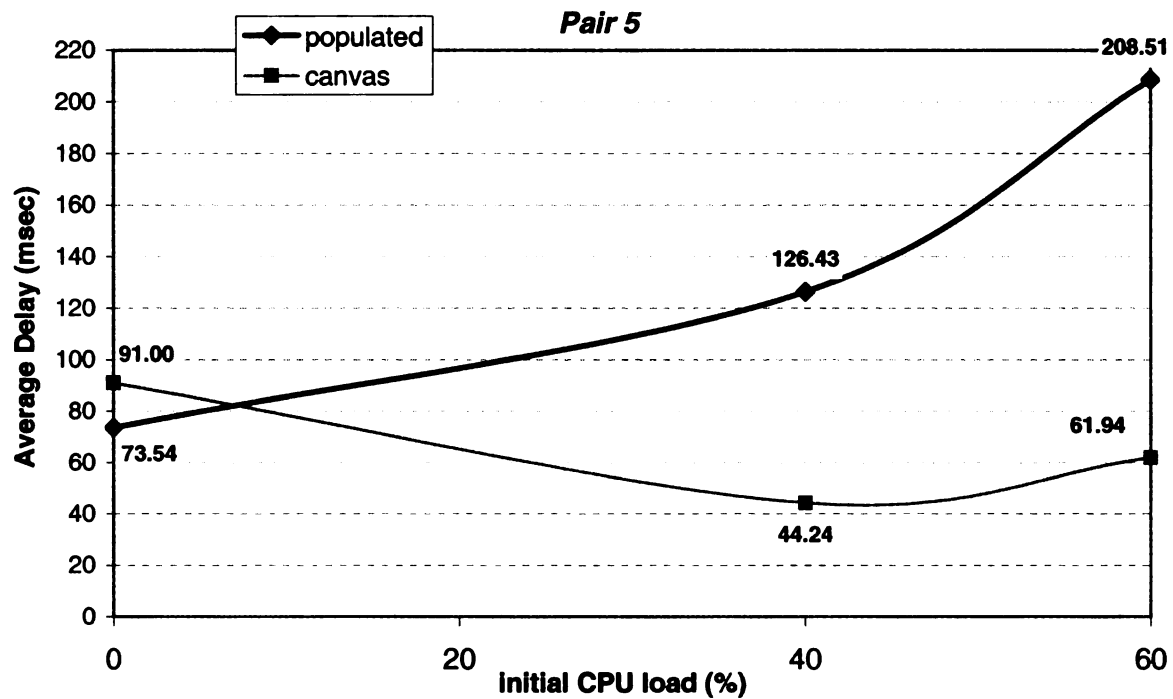


Figure 6.13: A sample of end-to-end service path delay as load increases: pairs 5 and 6 (continued on the next page).

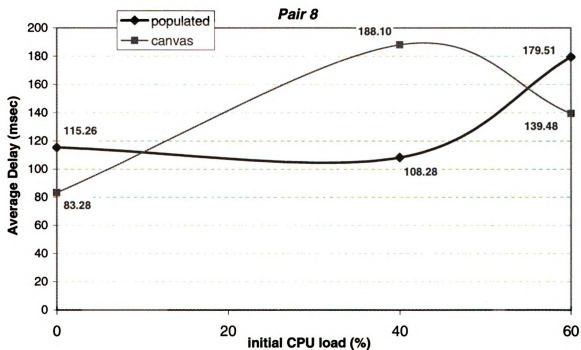
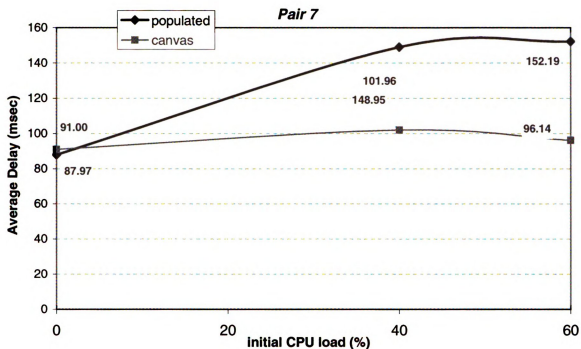


Figure 6.13: A sample of end-to-end service path delay as load increases: pairs 7 and 8 (continued on the next page).

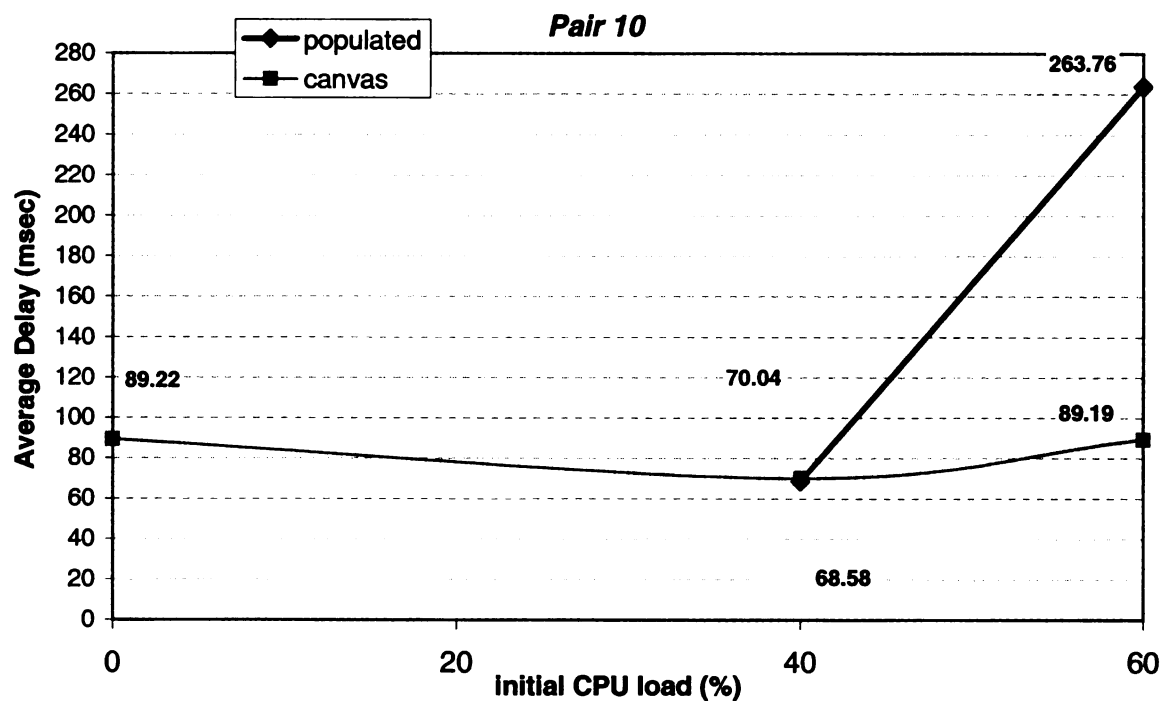
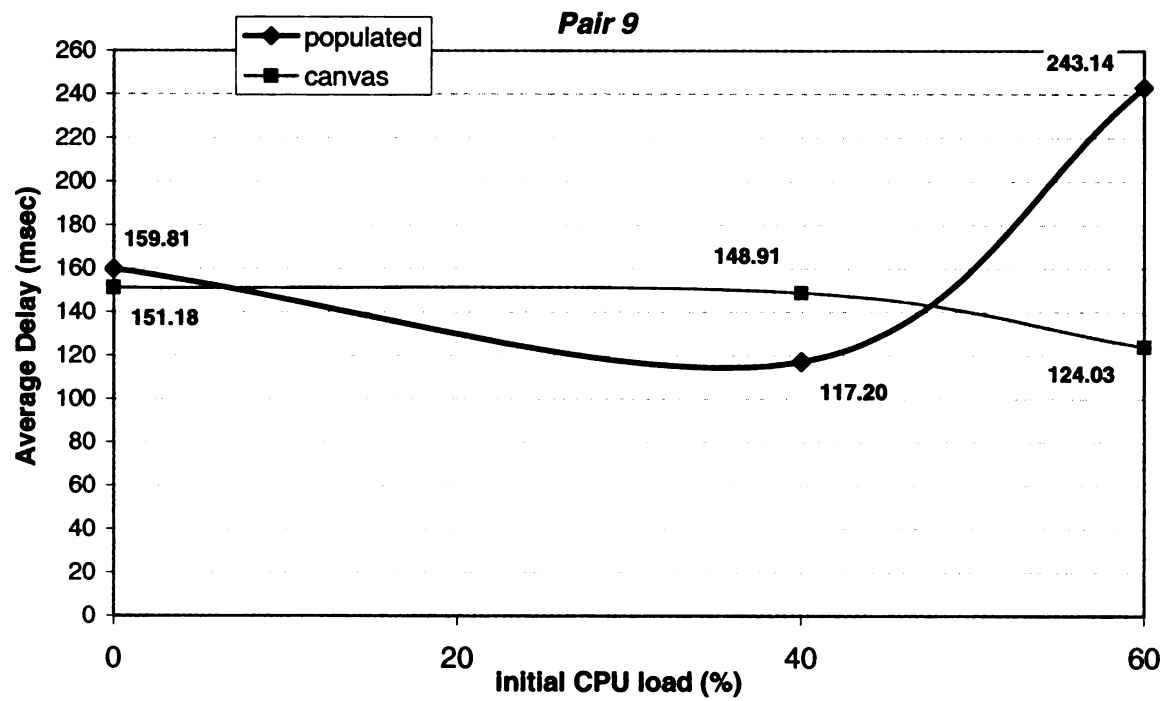


Figure 6.13: A sample of end-to-end service path delay as load increases: pairs 9 and 10.

with a smaller variance than that of the populated strategy. This is due to the fact that the canvas strategy considers only the availability of resources to select service paths, whereas the populated strategy is constrained by the availability of functions as well as resources. Figure 6.14 shows a sample of CPU load distribution on the overlay nodes under the two strategies, as the average load on nodes increases (missing data points have not been available in these particular samples). These plots show that the populated strategy drives a larger number of nodes to maximum load as compared to the canvas strategy. For example, after one service path has been established between nodes in each pair, with 0% initial load, 3 nodes reach the maximum capacity in the populated strategy, but no node reaches its maximum capacity in the canvas strategy. This number refers resource status at the end of run cycling of all pairs, when services have been established for all pairs, nodes may have reached their maximum capacity earlier. At 40% initial load, 16 nodes reach the maximum capacity in the populated strategy, while this is true of only 6 nodes in the canvas strategy. Finally, at 60% initial load, this number is 16 for the populated strategy and 13 for the canvas strategy. In summary, the canvas strategy performs better than the populated strategy in terms of resource consumption, since it is not constrained by the availability of functions executing in a subset of nodes.

6.5 Related Work

Overlay service composition [76, 80, 156, 157] is particularly useful in distributed processing of data streams [158]. A fundamental issue in overlay service composition is the probing mechanism used to locate and select a series of nodes on which to execute required

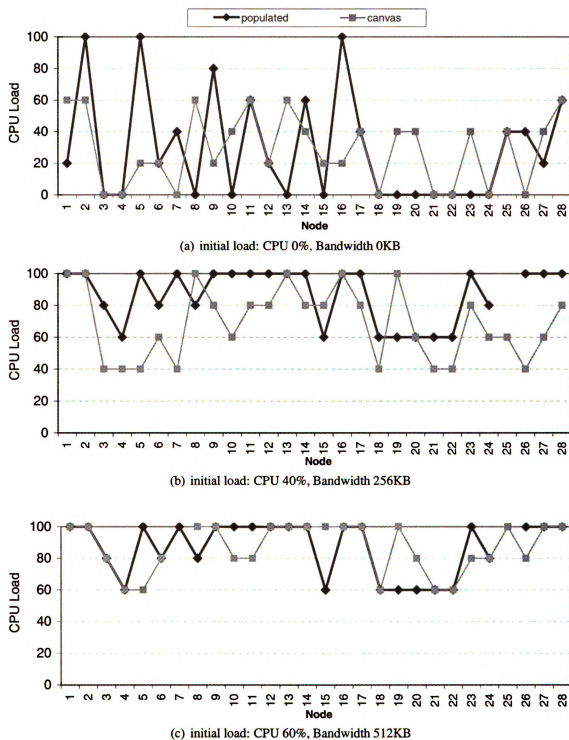


Figure 6.14: A sample load distribution on the 28 overlay nodes after service paths are setup for all 10 pairs of nodes.

stream processing operators (service elements). Composed services should consume distributed resources optimally, similarly to load-balancing and task assignment problems, while providing acceptable end-to-end quality. Researchers have investigated two main aspects of this problem: locating suitable nodes to execute stream processing operators [19, 152, 159–161] and sharing of services and processed data [153, 162]. Below, we discuss representative works; in both areas, though Dynamis focuses on the first aspect.

To consume distributed resources efficiently when composing services, a probing algorithm needs to consider sharing of services and processed data. Repantis et al. [153] proposed the Synergy composition framework for distributed stream processing. The main contribution of Synergy is that it provides an approach to efficiently reuse existing data streams and processing components when composing services. Moreover, it provides distributed algorithms to discover and evaluate the reusability of available data stream processing components that already generate the streams required by a service. This process includes estimating the impact of reuse on the quality of executing services, as well as assessing the end-to-end quality yielded if an existing entity is reused. Overall, the Synergy developers show that it improves resource utilization and QoS provisioning. Seshadri et al. [162] address the problem of optimizing multiple queries in distributed data stream systems, which is another dimension of operator reuse to optimize overall deployment. To execute distributed processing for a query, the system can produce or select different query plans, which define execution order of the operators. Their work investigates the relationship between query plans and placement of operators where multiple queries execute simultaneously. It provides distributed approaches that combine selection of query plans and operator placement to maximize the utility function of a distributed system.

In service composition, a probing algorithm has to locate suitable nodes to place distributed processing operators. Gu et al. [19] have introduced SpiderNet, a peer-to-peer service composition framework that performs distributed bounded probing, as well as proactive failure recovery to handle dynamic changes. A key property of the SpiderNet probing algorithm is that probes are distributed only to nodes that are capable of executing the required functions. A probe contains the function graph, resource requirements for each function, and expected QoS, along with the set of nodes it has traversed and their corresponding resource status. In distributed probing, upon receiving a probe, a node uses a distributed hash table to identify neighbor nodes capable of executing the next task in the function graph. If the node can meet requirements described by the probe, it adds its resource status to the probe and forwards the probes only to the neighbors that can execute the next task. Probing results in a collection of *service path* candidates, from which the one with the lowest load is selected at the endpoint. This approach composes high-quality, resource-efficient services in systems in which nodes are populated with required functions, each function replicated on a number of nodes in advance.

Pietzuch et al. [152] proposed SBON to address the basic problem of locating suitable overlay nodes to place stream processing operators on them. The SBON design is based on a “cost space,” a multi-dimensional metric space where the distance between nodes is an estimate of the cost of routing between them (in terms of desired measurements such as latency, bandwidth, and processing power). For other dimensions, it uses a scaling function to map metrics such as CPU load that can be measured locally onto the cost space. To locate a node on which to place an operator, SBON uses a heuristic that first explores the virtual cost space and finds a network region that is suitable with respect to latency. Next,

the algorithm contacts a number of nodes and queries their current status (resources and current executing operators), and maps the operator to the node with the lowest cost. In another related work, Liang and Nahrstedt [159] address the problem when data streams from multiple sources are processed and aggregated to be delivered to multiple destinations. They provide a heuristic central-based algorithm that can efficiently find low cost composition mapping in a large-scale network, evaluated by algorithmic analysis and simulation. They also show that the service composition problem is *NP* hard even in the case of a single source, single destination service.

The novelty of our work is that it addresses three major aspects of the probing problem simultaneously: introducing a generic optimization technique based on distributed selection, exploiting this technique to provide an extensible algorithmic framework, and conducting extensive experimental evaluation on the Internet. The proposed technique and framework are generic in that they can accommodate different probing mechanisms. In particular, the optimization technique presented here is not bound to any specific algorithm and can be applied to various (domain-specific) decentralized probing protocols such as SpiderNet [19] and SBON [152]. For example, the method does not rely on any assumption about pre-determined availability of functions at certain nodes [19]. Rather than evaluating Dynamis via simulation [159], our experimental evaluation realizes variants of existing probing algorithms and explores the effect of Dynamis in terms of overhead and quality of service composition on the Internet.

6.6 Summary

In this chapter, we described Dynamis, a probing algorithm to support composition of distributed overlay services. Dynamis is based on distributed service selection, which reduces the overhead of probing to locate suitable nodes on which to instantiate services. We presented the Dynamis algorithm and used it to empirically assess performance of different service composition strategies on the Internet. The experimental results show that using distributed selection reduces the probing overhead in service composition, while still finding high-quality service paths. Next, we turn back to the autonomic infrastructure itself, specifically, design techniques that can aid both in constructing and in maintaining the infrastructure.

Chapter 7

Conclusions and Future Directions

Our investigation addressed autonomic communication services based on an overlay network framework. We showed that an overlay-based infrastructure can combine and orchestrate dynamic software configuration, cross-layer coordination, and cross-platform collaboration to establish and adapt complex communication services. In particular, high-performance, high-quality data streaming, as well as distributed processing of data streams, require such an infrastructure to dynamically instantiate and configure necessary distributed service components, and automatically reconfigure those components as conditions change. The Service Clouds architecture realizes the building blocks and the interactions among them necessary to construct adaptive communication services. In this chapter, we briefly summarize our specific contributions, revisit the problem in light of lessons learned, and discuss possible topics for future study.

7.1 Contributions

This dissertation produced four main contributions summarized below.

- 1. Overlay-based architecture for autonomic communication.** We proposed an overlay-based architecture, Service Clouds, to support autonomic communication. This architecture identifies the main building blocks to compose autonomic services and can be used as a guideline to rapidly construct autonomic communication services.
- 2. High-performance communication substrate on the Internet.** We used Service Clouds to construct two high-performance communication services on an Internet-based testbed. The TCP-Relay service establishes overlay relays to expedite data transfer between two endpoints whenever possible, and the MCON service dynamically establishes high-quality overlay multi-path connection between two endpoints.
- 3. High-quality mobile computing substrate at the wireless edge.** We introduced Mobile Service Clouds and conducted three case studies that show how an overlay-based infrastructure can dynamically instantiate and configure proxies to deliver high-quality streaming services in wireless environments. In the first case study, we conducted high-quality streaming through a wireless channel by dynamically intercepting a stream at the operating system level, and applying error-correction on the stream within a proxy on a node at the wireless edge. In the second case study, we showed how run-time monitoring and migration of a proxy function delivers streams without interruption when the proxy crashes. Finally, we showed how dynamic instantiation of proxies supports multicasting at the wireless edge, as well as pervasive streaming for mobile users who move about different network domains.
- 4. Generic algorithmic framework for dynamic distributed service composition.** Dis-

tributed stream processing requires composition of a number of operators to process a stream. To locate nodes on which to configure or to instantiate operators, the system needs to perform probing, which can lead to significant traffic overhead. We introduced Dynamis, a generic optimization technique to reduce the overhead of distributed probing. Empirical results of applying Dynamis on different probing mechanisms, implemented within the Deep Service Clouds prototype, showed it can significantly reduce the probing overhead while establishing high-quality service paths.

Combined, these results support design and construction of autonomic systems that facilitate distributed services for emerging applications such as distributed processing of sensor data for large-scale environmental monitoring and critical infrastructure protection.

7.2 Problem Revisited

We believe it may be useful to step back and reconsider the problem of designing autonomic communication infrastructure, given the knowledge we have gained from conducting this research. In particular, we consider higher-level abstractions that may be helpful in designing such systems. This discussion is intended for researchers who may continue this line of study.

In general, adaptation can be *underlay-driven* or *request-driven*. This dissertation has focused primarily on the former. In underlay-driven adaptation, an automatic framework has to maintain the quality of a service by adapting the overlay service graph when the network conditions change. For example, in Figure 7.1, service element S_4 needs to be relocated from node l to node m when node l or link $g - l$ fails. Another example is when

a user moves and his/her connection is switched between different access domains. For instance, in Figure 7.1, when user $M1$ moves to a new network domain, the system needs to forward the stream to the new domain. One way to accomplish uninterrupted streaming is to have a proxy at the last hop (node l) that redirects the stream, as we demonstrated in Section 5.5 (Chapter 5).

In request-driven adaptation, the purpose of adaptation is to accommodate new service requests from users by better utilizing resources, for example, by increasing the level of sharing of the processed data. Request-driven adaptation is applicable when the system is requested to provide a service that *matches* an already established service. Two services match if they have at least one output in common among their distributed components. An example is shown in Figure 7.1. Assume user $M1$ requires a new stream processing configuration, that matches the current service graph except at the last service element S_4 . In this case, the system could replace S_4 by a new service, rather than composing a completely new service path from scratch. Replacement is viable if $M1$ is the only user of the executing service path, otherwise, the new service element can be added on node l (if the overlay node l can host the new service element). As another example, consider a user request for a service that matches a service already established for other users. For instance, assume user $M2$ joins in the same domain as user $M1$ and requests the same stream that is available from S_4 . In this case, the system could extend a replicate stream from S_4 towards $M2$. Finally, consider the previous example except that user $M3$ joins from a different domain. In this case an end-to-end overlay stretch property, such as delay or packet loss, may not be acceptable if user $M3$ receives an output from S_4 . However, if the system adapts by relocating S_4 to another node between $M1$ and $M3$ (such as node m),

then M1 and M3 may be able to share the S_4 output, while satisfying QoS requirements.

Figure 7.2 shows a possible model for self-managing streaming services, which could be used to support both underlay- and request-driven adaptation. In this model, the service *composer* coordinates instantiation and configuration of a set of distributed components to establish and maintain requested services. To locate nodes on which to instantiate components or to reconfigure existing components, the composer uses the *overlay algorithm* and *tracker*. The overlay algorithm implements probing mechanisms to map service components to an overlay network. The tracker maintains the registry of executing overlay service graphs. In addition, the tracker triggers service recomposition, whenever adaptation of an executing service graph is required.

Initially, an overlay network is not populated with any executing service graphs. An autonomic framework needs to support initial service setup by composing service graphs dynamically, in Figure 7.1, for example, composing a distributed stream processing service, that processes data from source $S1$ and delivers it to user $M1$. Later, the system needs to adapt executing components if conditions change. In addition to monitoring distributed services to conduct underlay-driven adaptation, to realize user-driven adaptation, the autonomic substrate includes a *registry* that keeps the records of composed service graphs. The registry enables the system to explore matching services, to share processed data, and to adapt executing service graphs to maximize such sharing while reducing probing overhead.

Let us next consider a basic building block for constructing overlay streaming service graphs, which we refer to as MetaRelay. The idea is that a streaming service graph could be reified by instantiating and configuring MetaRelays. Moreover, adapting a streaming service graph can also be realized by reconfiguring, inserting, and removing MetaRelays.

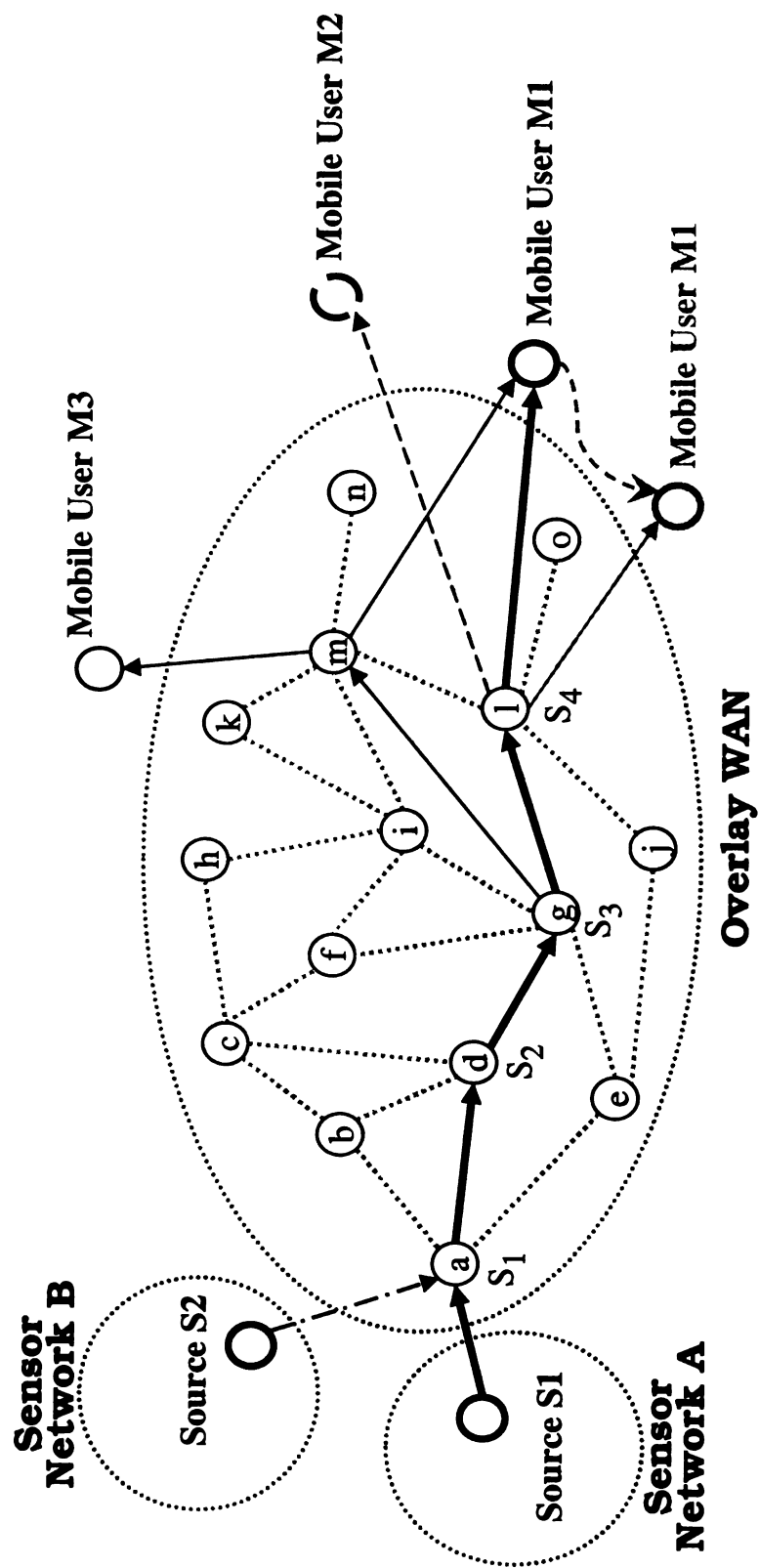


Figure 7.1: Illustration for adaptation scenarios.

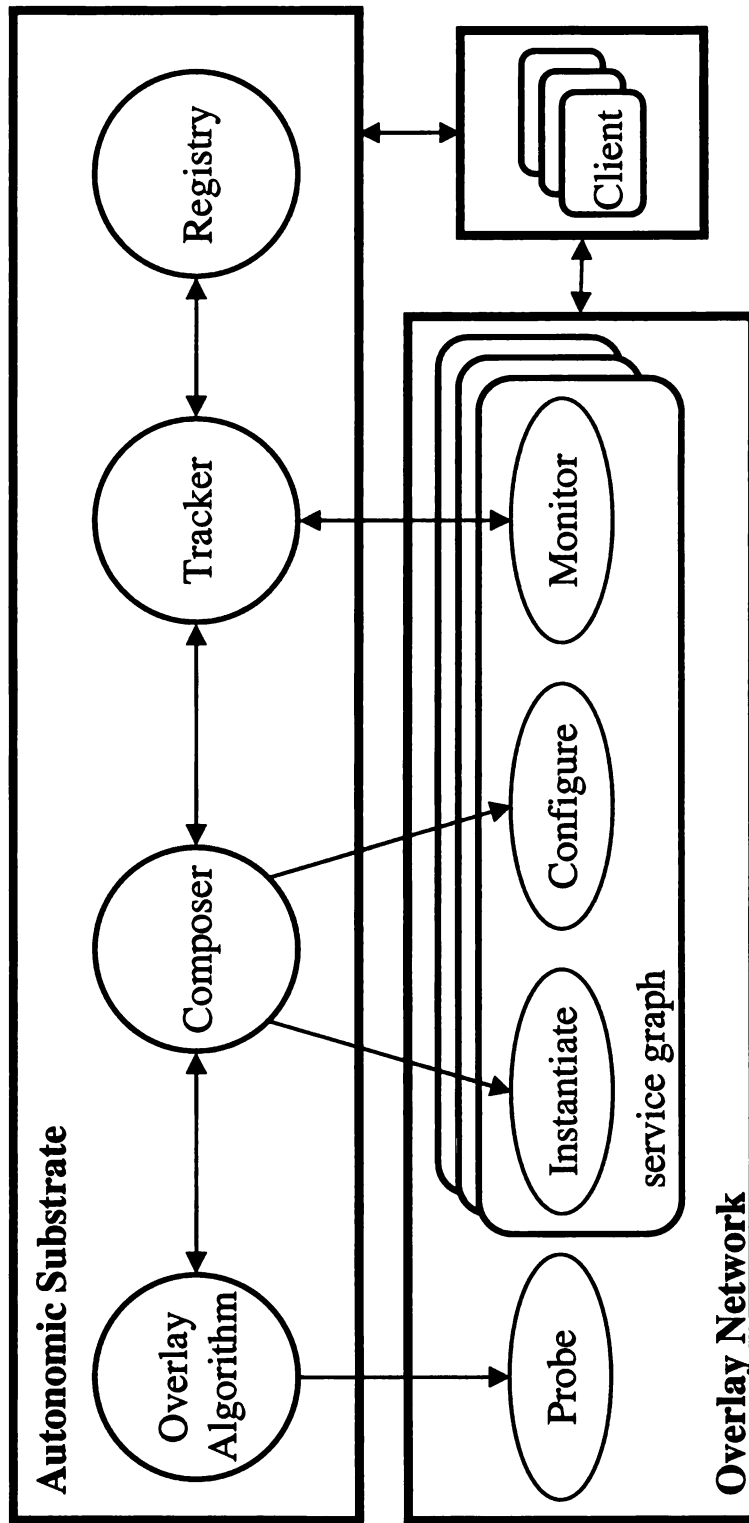


Figure 7.2: Basic model for self-managing services.

Figure 7.3 shows the main components of a MetaRelay. The *reflector* component is the data distributor; it accepts one or more stream inputs and sends out one or more streams (to local or remote components). The *transducer* component processes the content of a data stream (e.g., video transcoding, aggregation, FEC-encoding, and domain-specific filters). The *data sensor* component monitors the data stream (e.g., to measure transfer rate and packet loss rate). The *actuator* component analyzes the data sensor output and uses a simple rule-engine to actuate adaptation within the MetaRelay (e.g., inserting a decrypter when the incoming data turns to be encrypted), to notify another component to adapt (e.g., to request the video encoder at a source to lower bit rate when network congestion occurs), or to inform a service coordinator for higher-level decision-making (e.g., reporting transfer rate is lower than what has been specified).

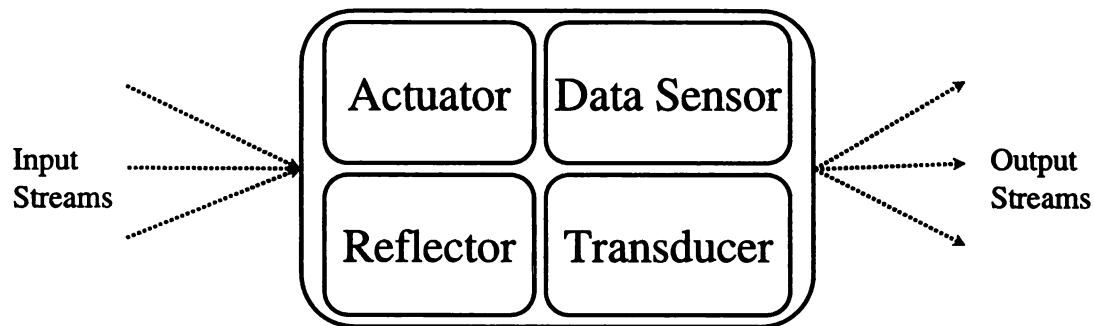


Figure 7.3: MetaRelay architecture.

Let us consider specific examples from previous chapters and describe them in terms of MetaRelay. Figure 7.4 shows a MetaRelay-based UDP relay used in routing packets in the MCON multipath-connection case study (Chapter 4). At the source and destination endpoints, the transducer adds and removes duplicate packets traversing different overlay paths. At intermediate overlay nodes, the reflector forwards the packets to the next hop.

Figure 7.5 shows an implementation of error-correction at a wireless link in a mobile

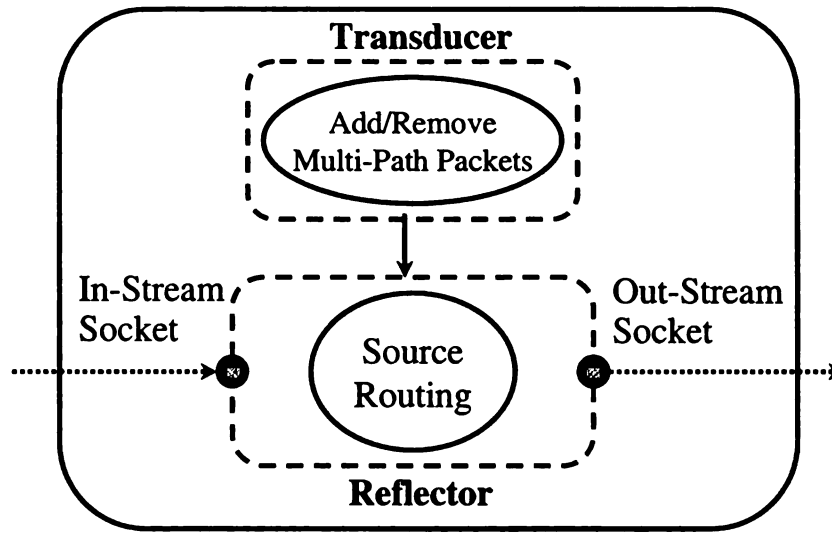


Figure 7.4: Multi-path connection relay modeled as MetaRelay.

service cloud (Chapter 5) based on the MetaRelay model. The receiving endpoint measures packet loss rate by checking the sequence numbers in the header of packets. Upon detecting intolerable loss rate, the actuator on the receiving endpoint announces this event to its counterpart on the sending endpoint. Accordingly, the actuator on the sending endpoint inserts an FEC encoder. When the data sensor of the MetaRelay at the receiving endpoint detects the encoded packets, the actuator inserts the counterpart FEC decoder.

Similarly, the MetaRelay abstraction supports modeling and implementing the seamless mobility case study, which we described in Chapter 5 (Figure 5.12). Figure 7.6 shows the instantiation of the seamless mobility study within our basic model for self-managing services. In this case, the MetaRelay on node *E1* consists of a reflector that accepts a UDP stream and multicasts it towards the users on wired subnet A. The autonomic substrate also registers the composed streaming service graph. Thus, when a second user (*M2*) joins subnet A and requests a stream which is already present in this subnet, the autonomic substrate connects the new user to the existing streaming service graph. When the mobile user

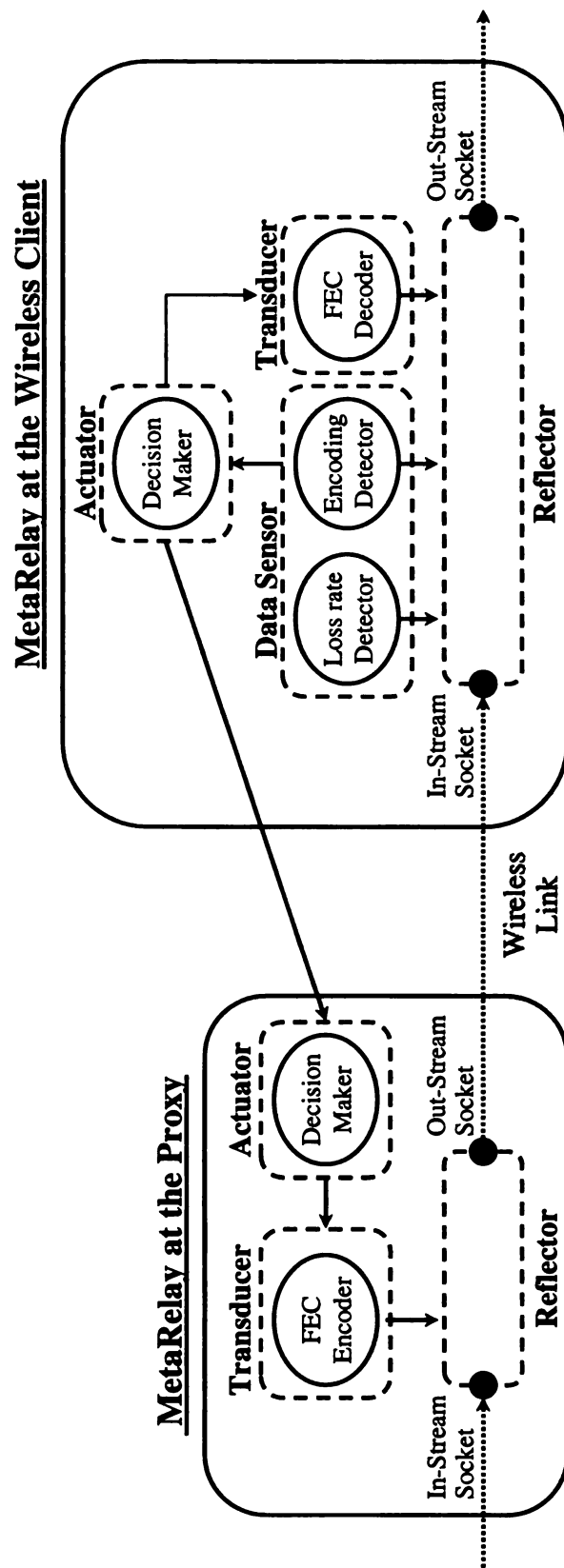


Figure 7.5: Wireless link error-correction modeled as MetaRelays.

M1 moves from wired subnet A to wireless subnet B, the client (which has been enabled to interact with the infrastructure) notifies the change of IP address to the tracker in the autonomic substrate (located on the Primary Proxy on node *N1* in this setup). Then, the autonomic substrate needs to adapt the overlay streaming service graph to send a copy of the stream to subnet B to which the user has moved. Consequently, the service composer instantiates a new MetaRelay on node *E2* to deliver the stream to wireless subnet B. The streaming in wireless subnet B is achieved according to the proxy-based error-correction model for wireless links described earlier. Afterwards, the composer uses the actuator on the MetaRelay on node *N1* in the deep service cloud to send an outgoing copy of the stream towards the newly instantiated MetaRelay on node *E2*. Other examples for MetaRelay include construction and maintenance of overlay multicast trees and the network of components used to process data gleaned from sensor networks.

We surmise that higher-level abstractions such as MetaRelays may be very useful in designing and maintaining autonomic communication infrastructure. As shown above, the MetaRelay can model services studied in earlier chapters. Moreover, such abstractions may be helpful in developing design patterns for autonomic software, a first step toward more rigorous software engineering of such systems.

7.3 Future Directions

The work presented in this dissertation can lead to several future research projects. We identify three major research directions that complement this dissertation.

Model-driven engineering of autonomic services. Applying rigorous software en-

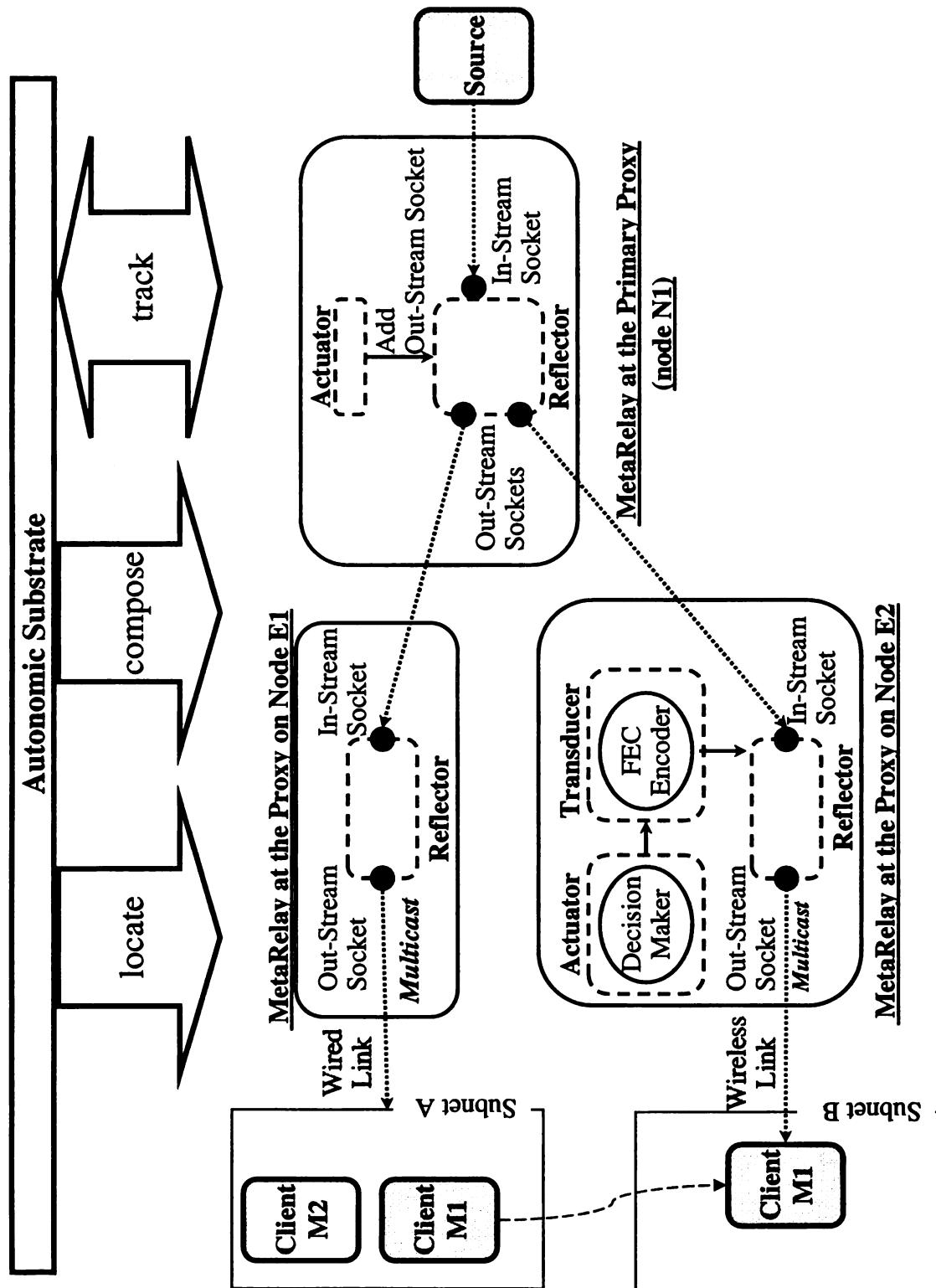


Figure 7.6: Seamless mobility case study modeled using MetaRelays.

gineering of autonomic software [163, 164] can produce systems that are more assured, and hence more reliable, than those developed with ad hoc techniques. Such an approach also supports automation in service construction via techniques such as code generation. In addition, formal modeling of the infrastructure enables automated inspection of the correctness and testing the operation of implementations. Modeling the three major parts of such systems (monitoring, decision making, response) and their interaction can lead to more secure and robust systems than those existing today.

Incorporation into the underlying network infrastructures. As networking infrastructures improve, they are expected to deliver more advanced and complicated services. While the core of the Service Clouds infrastructure is overlay-based, many parts of it can be incorporated into the underlying network infrastructure. Doing so would enable developers of network services to use advanced features to rapidly construct robust, high-performance enterprise solutions. For example, Cisco has recently introduced a service-oriented network architecture [165] to unify network-based services on a virtual resource plane. The results of the investigation in this dissertation can be used to support such trends in industry. In addition, as lower-level networking services provide more advanced features, the cross-layer coordination in the Service Clouds model can be enhanced to accomplish complex functions.

Real-time processing and streaming of high-volume data. High-volume data, such as those collected from sensor networks and simulation clusters, need to be processed and delivered to users on-demand. The Service Clouds infrastructure provides the framework and building blocks to implement distributed processing of data streams. Moreover, the Dynamis algorithm, that maps service elements into overlay nodes, can be expanded to

support sharing of processed data. These facilities can be used to advance a number of scientific disciplines that involve processing of streaming data. For instance, ecosystem monitoring generates high volume data around the clock [148]. Even a small-scale ecosystem monitoring network at Michigan State University, which currently consists of only 13 sensors and is being expanded, collects about 2MB of data every 30 minutes. Figure 7.7 is an example of stream processing in ecosystem monitoring, including noise filtering, motif extraction, and data mining of motif extraction [166]. Motif extraction deals with glean- ing events of interest from continuous streams. Data mining of motifs enables researchers to perform meaningful analysis of data. For instance, analyzing population of a specific type of birds to count them, study their singing patterns, and examine effects of human disturbance on their behaviors and populations.

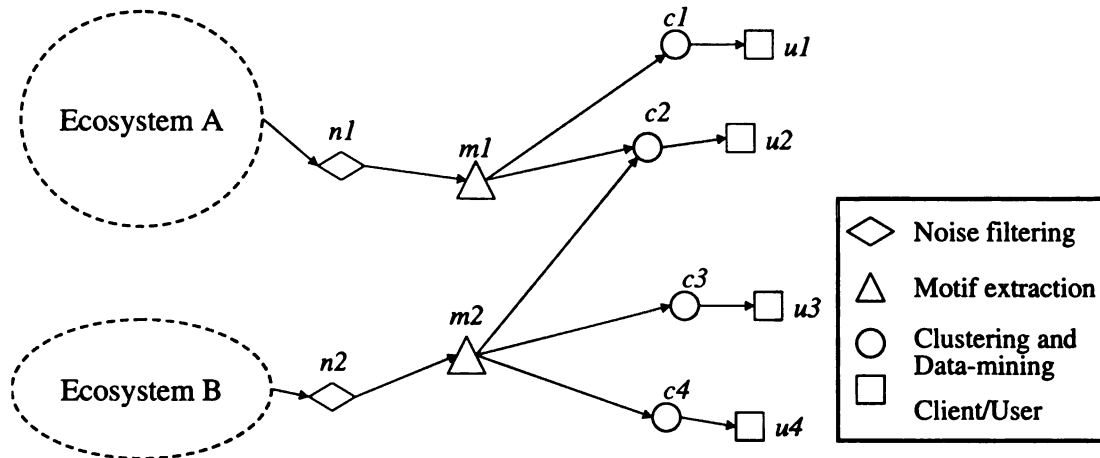


Figure 7.7: Ecosystem stream processing example.

Moreover, combining data from different areas allows researches to study ecosystems at a larger scale. For example, monitoring migration patterns and biodiversity can lead to a better understanding of ecosystem changes throughout a country, continent, or the planet. Our work on Service Clouds is a step toward exploiting advances in computing technology to automate the processing of such data as it is collected.

BIBLIOGRAPHY

Bibliography

- [1] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.
- [2] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3):181–187, 2005.
- [3] Philip K. McKinley, S. Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.
- [4] Mark Weiser. Ubiquitous computing. *IEEE Computer*, 26(10):71–72, October 1993.
- [5] Gordon Blair, Geoff Coulson, and Nigel Davies. Adaptive middleware for mobile multimedia applications. In *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 259–273, 1997.
- [6] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, April 1998.
- [7] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 121–143, April 2000.
- [8] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.
- [9] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Fabio Costa Michael Clarke, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikosk. The design and implementation of Open ORB 2. *IEEE DS Online, Special Issue on Reflective Middleware*, 2(6), 2001.
- [10] Roberto Baldoni, Carlo Marchetti, and Alessandro Termini. Active software replication through a three-tier approach. In *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, pages 109–118, Osaka, Japan, October 2002.
- [11] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, volume 2374 of *LNCS*, pages 205–230, June 2002.

- [12] Brian D. Noble, Mahadev Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, October 1997.
- [13] Sarita V. Adve, Albert F. Harris, Christopher J. Hughes, Douglas L. Jones, Robin H. Kravets, Klara Nahrstedt, Daniel Grobe Sachs, Ruchira Sasanka, Jayanth Srinivasan, and Wanghong Yuan. The Illinois GRACE Project: Global Resource Adaptation through CoopEration. In *Proceedings of the Workshop on Self-Healing, Adaptive, and self-MANaged Systems (SHAMAN)*, June 2002.
- [14] Christian Poellabauer, Hasan Abbasi, and Karsten Schwan. Cooperative run-time management of adaptive applications and distributed resources. In *Proceedings of the 10th ACM Multimedia Conference*, pages 402–411, France, December 2002.
- [15] Carla Ellis. The case for higher level power management. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, page 162. IEEE Computer Society, March 1999.
- [16] Heng Zeng, Carla Ellis, Alvin Lebeck, , and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, San Jose, California, October 2002.
- [17] Jiantao Kong and Karsten Schwan. KStreams: kernel support for efficient data streaming in proxy servers. In *Proceedings of the 15th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 159–164. ACM, 2005.
- [18] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 131–145, October 2001.
- [19] Xiaohui Gu, Klara Nahrstedt, and Bin Yu. SpiderNet: An integrated peer-to-peer service composition framework. In *Proceedings of IEEE International Symposium on High-Performance Distributed Computing (HPDC-13)*, pages 110–119, Honolulu, Hawaii, June 2004.
- [20] Paul Grace, Geoff Coulson, Gordon Blair, Laurent Mathy, David Duce, Chris Cooper, Wai Kit Yeung, and Wei Cai. GRIDKIT: Pluggable overlay networks for Grid computing. In *Proceedings of International Symposium on Distributed Objects and Applications(DOA)*, pages 1463–1481, Larnaca, Cyprus, October 2004.
- [21] Baochun Li, Jiang Guo, and Mea Wang. iOverlay: A lightweight middleware infrastructure for overlay application implementations. In *Proceedings of the Fifth ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *LNCS*, pages 135–154, Toronto, Canada, October 2004.
- [22] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostic, and Amin Vahdat. Mace-don: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the USENIX/ACM First Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 267–280, San Francisco, California, March 2004.

- [23] Steven D. Gribble, Matt Welsh, J. Robert von Behren, Eric A. Brewer, David E. Culler, N. Borisov, Steven E. Czerwinski, Ramakrishna Gummadi, Jon R. Hill, Anthony D. Joseph, Randy H. Katz, Z. M. Mao, S. Ross, and Ben Y. Zhao. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.
- [24] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijak Karamcheti. CANS: Composable and adaptive network services infrastructure. In *The 3rd USENIX Symposium on Internet Technology and Systems*, San Francisco, California, March 2001.
- [25] Farshad A. Samimi, Philip K. McKinley, S. Masoud Sadjadi, Chiping Tang, Jonathan K. Shapiro, and Zhinan Zhou. Service Clouds: Distributed infrastructure for adaptive communication services. *IEEE Transactions on Network and Service Management (TNSM)*, Special Issue on Self-Managed Networks, Systems and Services, 2007. In press.
- [26] Philip K. McKinley, Farshad A. Samimi, Jonathan K. Shapiro, and Chiping Tang. Service Clouds: A distributed infrastructure for constructing autonomic communication services. In *Proceedings of the 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC'06)*, pages 341–348, Indianapolis, Indiana, USA, September 2006. IEEE Computer Society.
- [27] Farshad A. Samimi, Philip K. McKinley, and S. Masoud Sadjadi. Mobile Service Clouds: A self-managing infrastructure for autonomic mobile computing services. In *Proceedings of the Second International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006)*, volume 3996 of LNCS, pages 130–141, Dublin, Ireland, June 2006. Springer-Verlag.
- [28] Farshad A. Samimi, Philip K. McKinley, S. Masoud Sadjadi, and Peng Ge. Kernel-middleware interaction to support adaptation in pervasive computing environments. In *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-Hoc Computing*, pages 140–145, Toronto, Ontario, Canada, October 2004. ACM Press.
- [29] Farshad A. Samimi and Philip K. McKinley. Dynamis: Dynamic overlay service composition for distributed stream processing. Technical Report MSU-CSE-06-39, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, December 2006. Shorter version submitted for conference publication.
- [30] Richard Hayton and Andrew Herbert. FlexiNet - a flexible component-oriented middleware system. In *Advances in Distributed Systems*, volume 1752 of LNCS, pages 497–508. Springer-Verlag, 2000.
- [31] Thomas Ledoux. OpenCorba: A reflective open broker. *Lecture Notes in Computer Science*, 1616:197–214, July 1999.
- [32] Philip K. McKinley, Udiyan I. Padmanabhan, and Nandagopal Ancha. Experiments in composing proxy audio services for mobile users. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 99–120, Heidelberg, Germany, November 2001.
- [33] Richard E. Schantz. Quorum distributed object integration (QuOIN) final report. BBN Technologies (website), 2002. <http://www.dist-systems.bbn.com/>.

- [34] Partha P. Pal, Franklin Webber, Richard E. Schantz, Michael Atighetchi, and Josef P. Loyall. Defense-enabling using advanced middleware: An example. In *Milcom 2001*, pages 92–101, Tysons Corner, Virginia, October 2001.
- [35] Douglas C. Schmidt, Rick Schantz, Mike Masters, Joseph Cross, David Sharp, and Lou DiPalma. Toward adaptive and reflective middleware for network-centric combat systems. *CrossTalk - The Journal of Defense Software Engineering*, pages 10–16, November 2001.
- [36] Oguz Angin, Andrew T. Campbell, Michael E. Kounavis, and Raymond R.-F. Liao. The Mobeware toolkit: Programmable support for adaptive mobile networking. *IEEE Personal Communications Magazine, Special Issue on Adapting to Network and Client Variability*, 5(4):32–43, 1998.
- [37] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, May 2004.
- [38] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.
- [39] Andrew T. Campbell, Geoff Coulson, and Michael E. Kounavis. Managing complexity: Middleware explained. *IT Professional, IEEE Computer Society*, 1(5):22–28, September/October 1999.
- [40] Pattie Maes. Concepts and experiments in computational reflection. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 147–155. ACM Press, 1987.
- [41] Fabio M. Costa, Hector A. Duran, Nikos Parlavantzas, Katia B. Saikoski, Gordon S. Blair, and Geoff Coulson. The role of reflective middleware in supporting the engineering of dynamic applications. In *Reflection and Software Engineering*, volume 1826 of *LNCS*, pages 79–98. Springer-Verlag, 2000.
- [42] Louise Moser, P.Michael Melliar-Smith, Priya Narasimhan, L.A. Tewksbury, and Vana Kalogeraki. The Eternal system: An architecture for enterprise applications. In *Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC'99)*, pages 214–222, July 1999.
- [43] S. Masoud Sadjadi and Philip K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, March 2004.
- [44] Pierre Charles David, Thomas Ledoux, and Noury M. N. Bouraqadi-Saadani. Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, October 2001.
- [45] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming*. Addison Wesley, 2000.
- [46] OMG. The OMG's CORBA Website. <http://www.corba.org/>.

- [47] Gordon S. Blair, Geoff Coulson, Philippe Robin, and Michael Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 191–206, Lake District, UK, 1998.
- [48] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.
- [49] Douglas C. Schmidt and Chris Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communication Magazine Special Issue on Design Patterns*, 37(4):54–63, 1999.
- [50] Manuel Román, Dennis Mickunas, Fabio Kon, and Roy H. Campbell. LegORB and Ubiquitous CORBA. In *Proceedings of the IFIP/ACM Middleware Workshop on Reflective Middleware*, pages 1–2, Palisades, NY, April 2000.
- [51] Fabio Kon, Roy Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A distributed operating system for dynamic heterogeneous environments. *9th IEEE International Symposium on High Performance Distributed Computing*, August 2000.
- [52] Carlo Marchetti, Luigi Verde, and Roberto Baldoni. CORBA request portable interceptors: A performance analysis. In *the 3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, pages 208–217, Rome, Italy, September 2001.
- [53] Mary Kirtland. Object-oriented software development made simple with COM+ runtime services. *Microsoft Systems Journal*, 12(11):49–59, November 1997.
- [54] Priya Narasimban, Louise E. Moser, and P.Michael Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, 32(7):62–68, July 1999.
- [55] Christian Poellabauer and Karsten Schwan. Kernel support for the event-based cooperation of distributed resource managers. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*, pages 3–12, San Jose, California, September 2002.
- [56] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 48–63, December 1999.
- [57] Wanghong Yuan, Klara Nahrstedt, Sarita V. Adve, Douglas L. Jones, and Robin H. Kravets. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. In *Proceedings of the SPIE/ACM Multimedia Computing and Networking Conference (MMCN '03)*, Santa Clara, California, January 2003.
- [58] Farshad A. Samimi and Philip K. McKinley. A survey of kernel-middleware interaction in support of cross-layer adaptation in mobile computing. Technical Report MSU-CSE-04-44, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, October 2004.
- [59] Kostas G. Anagnostakis, Sotiris Ioannidis, Stefan Miltchev, John Ioannidis, and Jonathan M. Smith. Efficient packet monitoring for network management. In *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2002.

- [60] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX 1998 Annual Technical Conference*, pages 39–52, June 1998. <http://now.cs.berkeley.edu/Slic/>.
- [61] Terrence Mitchem, Raymond Lu, and Richard O'Brien. Using kernel Hypervisors to secure applications. *Annual Computer Security Application Conference (ACSAC '97)*, December 1997. <http://www.securecomputing.com/khyper/>.
- [62] Jacco Taal, Ivaylo Haratcherev, Koen Langendoen, and Inald Lagendijk. Quality of service controlled adaptive video coding over IEEE 802.11 wireless links. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME'03)*, volume 1, pages 189–192, Baltimore, Maryland, July 2003.
- [63] Christian Poellabauer, Karsten Schwan, and Richard West. Lightweight kernel/user communication for real-time and multimedia applications. In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSS-DAV 2001)*, pages 145–154, Port Jefferson, NY, June 2001.
- [64] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. Application-aware adaptation for mobile computing. *Operating Systems Review*, 29(1):52–55, 1995.
- [65] Brian Noble. System support for mobile, adaptive applications. *IEEE Personal Communications*, 7(1):44–49, February 2000.
- [66] Distributed Extensible Open Systems (the DEOS project). <http://www.cc.gatech.edu/systems/projects/DEOS/>. Georgia Institute of Technology - College of Computing.
- [67] Yang hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS*, pages 1–12, June 2000.
- [68] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM*, pages 205–217, Pittsburgh, Pennsylvania, August 2002.
- [69] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 329–350, November 2001.
- [70] Ben Y. Zhao, John D. Kubiawicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD 01/1141, UC Berkeley Computer Science Division, April 2001.
- [71] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, San Diego, California, August 2001.
- [72] Mark Yarvis, Peter L. Reiher, and Gerald J. Popek. Conductor: A framework for distributed adaptation. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 44–49, Rio Rico, Arizona, March 1999.

- [73] Stefan Savage, Tom Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. Detour: a case for informed Internet routing and transport. *IEEE Micro*, 9(1):50–59, January 1999.
- [74] Bhaskaran Raman, Sharad Agarwal, Yan Chen, Matthew Caesar, Weidong Cui, Per Johansson, Kevin Lai, Tal Lavian, Sridhar Machiraju, Z. Morley Mao, George Porter, Timothy Roscoe, and Mukund. The SAHARA model for service composition across multiple providers. In *Proceedings of the First International Conference on Pervasive Computing*, volume 2414 of *LNCS*, pages 1–14, Zurich, Switzerland, August 2002. Springer Berlin / Heidelberg.
- [75] Jingwen Jin and Klara Nahrstedt. QoS service routing in one-to-one and one-to-many scenarios in next-generation service-oriented networks. In *Proceedings of The 23rd IEEE International Performance Computing and Communications Conference (IPCCC2004)*, Phoenix, Arizona, April 2004.
- [76] Xiaodong Fu and Vijay Karamcheti. Automatic creation and reconfiguration of network-aware service access paths. *Computer Communications*, 28(6):591–608, 2005.
- [77] Geoff Coulson, Paul Grace, Gordon Blair, Wei Cai, Chris Cooper, Daveid Duce, Laurent Mathy, , Wai Kit Yeung, Barry Porter, Musbah Sagar, and Wei Li. A component-based middleware framework for configurable and reconfigurable Grid computing. *Concurrency and Computation: Practice and Experience*, 18(8):865–874, 2005.
- [78] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Resource-aware distributed stream management using dynamic overlays. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS 2005)*, pages 783–792, Columbus, OH, USA, June 2005. IEEE Computer Society.
- [79] Baochun Li, Dongyan Xu, and Klara Nahrstedt. An integrated runtime QoS-aware middleware framework for distributed multimedia applications. *Multimedia Systems*, 8(5):420–430, 2002.
- [80] Dongyan Xu and Xuxian Jiang. Towards an integrated multimedia service hosting overlay. In *Proceedings of ACM Multimedia 2004*, pages 96–103, New York, NY, October 2004. ACM Press.
- [81] Xiaohui Gu, Klara Nahrstedt, Rong N. Chang, and Christopher Ward. QoS-assured service composition in managed service overlay networks. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS 2003)*, Providence, Rhode Island, May 2003.
- [82] Nelson R. Manohar, Ashish Mehra, Marc H. Willebeek-LeMair, and Mahmoud Naghshineh. A framework for programmable overlay multimedia networks. *IBM Journal of Research and Development*, 43(4):555–577, July 1999.
- [83] John W. Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed content delivery across adaptive overlay networks. *IEEE/ACM Transactions on Networking (TON)*, 12(5):767–780, October 2004.

- [84] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. *Lecture Notes in Computer Science*, 2218:160–178, 2001.
- [85] Xiaohui Gu, Philip S. Yu, and Klara Nahrstedt. Optimal component composition for scalable stream processing. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS 2005)*, pages 773–782, Columbus, OH, USA, June 2005. IEEE Computer Society.
- [86] Jeffrey O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 15–22, St. Louis, Missouri, 2005.
- [87] S. Masoud Sadjadi. *Transparent shaping of existing software to support pervasive and autonomic computing*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 2004.
- [88] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6):43–48, June 2002.
- [89] S. Masoud Sadjadi, Philip K. McKinley, and Betty H.C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *Proceedings of the first Workshop on the Design and Evolution of Autonomic Application Software 2005 (DEAS'05), in conjunction with ICSE 2005*, St. Louis, Missouri, May 2005.
- [90] S. M. Sadjadi, Philip K. McKinley, Eric P. Kasten, and Zhinan Zhou. MetaSockets: Design and operation of run-time reconfigurable communication services. *Software: Practice and Experience (SP&E). Special Issue: Experiences with Auto-adaptive and Reconfigurable Systems.*, 36:1157–1178, 2006.
- [91] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.
- [92] W3C. *Document Object Model (DOM) Level 2 Core Specification*, version 1.0 edition, November 2000.
- [93] SAX. Simple API for XML. <http://www.saxproject.org/>.
- [94] W3C. XML binary characterization working group public page. <http://www.w3.org/XML/Binary/>.
- [95] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proceedings of ACM Workshop on Hot Topics in Networks (HotNets-I)*, pages 59–64, Princeton, New Jersey, October 2002. <http://www.planet-lab.org/>.
- [96] Sun Microsystems, Inc. *Java™ Message Service Specification*, version 1.1 edition, April 2002.
- [97] Zhinan Zhou. *Design and evaluation of adaptive software for mobile computing systems*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 2006.

- [98] The RAPIDware Project. <http://www.cse.msu.edu/rapidware>. Software Engineering and Network Systems Laboratory, Michigan State University - Department of Computer Science and Engineering, East Lansing, MI, USA.
- [99] Martin Swany. Improving throughput for Grid applications with network logistics. In *Proceedings of IEEE/ACM Conference on High Performance Computing and Networking*, pages 23–34, November 2004.
- [100] Yong Liu, Yu Gu, Honggang Zhang, Weibo Gong, and Don Towsley. Application level relay for high-bandwidth data transport. In *Proceedings of the First Workshop on Networks for Grid Applications (GridNets)*, San Jose, California, October 2004.
- [101] Himabindu Pucha and Y. Charlie Hu. Overlay TCP: Multi-hop overlay transport for high throughput transfers in the Internet. Technical Report TR-05-08, Dept. Electrical and Computer Engineering, Purdue University, March 2005.
- [102] Matthew Mathis, Jeffrey Semke, Jamshid Madhavi, and Teunis Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM Computer Communications Review*, 27(3):67–82, 1997.
- [103] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: a simple model and its empirical validation. *IEEE/ACM Transactions on Networking*, 8(2):133–145, 2000.
- [104] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Transactions on Networking*, 11(4):537–549, 2003.
- [105] Qi He, Constantinos Dovrolis, and Mostafa Ammar. On the predictability of large transfer TCP throughput. In *Proceedings of the ACM SIGCOMM*, Philadelphia, PA, August 2005.
- [106] Chiping Tang and Philip K. McKinley. Improving multipath reliability in topology-aware overlay networks. In *Proceedings of the Fourth International Workshop on Assurance in Distributed Systems and Networks (ADSN), held in conjunction with the 25th IEEE International Conference on Distributed Computing Systems*, Columbus, Ohio, June 2005.
- [107] C. Tang and P. K. McKinley. On the cost-quality tradeoff in topology-aware overlay path probing. In *Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP)*, pages 268–279, Atlanta, Georgia, November 2003.
- [108] Joseph Ishac and Mark Allman. On the performance of TCP spoofing in satellite networks. In *Proceedings of the IEEE Military Communications Conference (MILCOM 2001)*, pages 701–705, October 2001.
- [109] Michele Luglio, M. Yahya Sanadidi, Mario Gerla, and James Stepanek. On-board satellite "Split TCP" proxy. *IEEE Journal on Selected Areas in Communications*, 22(2), 2003.
- [110] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the ACM SIGCOMM*, pages 89–102, October 2002.
- [111] Sally Floyd. Highspeed TCP for large congestion windows. Technical Report RFC 3649, IETF, 2003.

- [112] Cheng Jin, David X. Wei, and Steven H. Low. Fast TCP: Motivation, architecture, algorithms, performance. In *Proceedings of the IEEE INFOCOM*, pages 2490–2501, March 2004.
- [113] J. Suurballe and R. Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14:325–336, 1984.
- [114] Johnny Chen. *New Approaches to Routing for Large-Scale Data Networks*. PhD thesis, Rice University, Houston, Texas, June 1999.
- [115] Srihari Nelakuditi and Zhi-Li Zhang. On selection of paths for multipath routing. *Lecture Notes in Computer Science (IWQoS 2001)*, 2092, 2001.
- [116] Srinivas Vutukury and J.J. Garcia-Luna-Aceves. An algorithm for multipath computation using distance-vectors with predecessor information. In *Proceedings of the IEEE International Conference on Computer Communications and Networks (ICCCN)*, pages 534–539, October 1999.
- [117] Deepinder Sidhu, Raj Nair, and Shukri Abdallah. Finding disjoint paths in networks. In *Proceedings of ACM SIGCOMM*, pages 43–51, September 1991.
- [118] Srinivas Vutukury. *Multipath Routing Mechanisms For Traffic Engineering And Quality Of Service In The Internet*. PhD thesis, University of California, Santa Cruz, 2001.
- [119] Anthony Young, Jiang Chen, Zheng Ma, Arvind Krishnamurthy, and Randolph Y. Wang. Overlay mesh construction using interleaved spanning trees. In *Proceedings of INFOCOM*, March 2004.
- [120] Ming Zhang, Junwen Lai, Arvind Krishnamurthy, Larry Peterson, and Randolph Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proceedings of USENIX Annual Technical Conference*, pages 99–112, 2004.
- [121] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Anthony D. Joseph, and John D. Kubiatowicz. Exploiting routing redundancy via structured peer-to-peer overlays. In *Proceedings of ICNP*, pages 246–257, November 2003.
- [122] Junghee Han and Farnam Jahanian. Impact of path diversity on multi-homed and overlay networks. In *Proceedings of International Conference on Dependable Systems and Networks (DSN'04)*, pages 29–38, June 2004.
- [123] Weidong Cui, Ion Stoica, and Randy H. Katz. Backup path allocation based on a correlated link failure probability model in overlay networks. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP'02)*, pages 236–245, November 2002.
- [124] Minseok Kwon and Sonia Fahmy. Topology-aware overlay networks for group communication. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2002)*, pages 127–136, Florida, USA, May 2002.
- [125] Yan Chen, David Bindel, Hanhee Song, and Randy Katz. An algebraic approach to practical and scalable overlay network monitoring. In *Proceedings of SIGCOMM*, pages 55–66, Portland, Oregon, 2004.

- [126] Aki Nakao, Larry Peterson, and Andy Bavier. A routing underlay for overlay networks. In *Proceedings of ACM SIGCOMM*, pages 11–18, Karlsruhe, Germany, August 2003.
- [127] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. *IEEE Personal Communications*, 5(4):10–19, August 1998.
- [128] Johan Kristiansson and Peter Parnes. An application-layer approach to seamless mobile multimedia communication. *IEEE eTransactions on Network and Service Management (eTNSM)*, 3(1):33–42, 2006.
- [129] Bruce Zenel. A general purpose proxy filtering mechanism applied to the mobile environment. *Wireless Networks*, 5:391–409, 1999.
- [130] Yongjie Zheng. MobiGATE: A mobile computing middleware for the active deployment of transport services. *IEEE Transactions on Software Engineering*, 32(1):35–50, 2006.
- [131] Mema Roussopoulos, Petros Maniatis, Edward Swierk, Kevin Lai, Guido Appenzeller, and Mary Bake. Person-level routing in the mobile people architecture. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems*, pages 165–176, Boulder, Colorado, October 1999.
- [132] Philip K. McKinley, Chiping Tang, and Arun P. Mani. A study of adaptive forward error correction for for wireless collaborative computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):936–947, September 2002.
- [133] Philip K. McKinley, Udiyan I. Padmanabhan, Nandagopal Ancha, and S. Masoud Sadjadi. Composable proxy services to support collaboration on the mobile Internet. *IEEE Transactions on Computers (Special Issue on Wireless Internet)*, pages 713–726, June 2003.
- [134] Zhinan Zhou, Philip K. McKinley, and S. Masoud Sadjadi. On quality-of-service and energy consumption tradeoffs in FEC-enabled audio streaming. In *Proceedings of the 12th IEEE International Workshop on Quality of Service (IWQoS 2004)*, Montreal, Canada, June 2004. Selected as the best student paper.
- [135] Peng Ge. *Interactive Video Multicast in Wireless LANs*. PhD thesis, Michigan State University, Department of Computer Science and Engineering, December 2004.
- [136] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, April 1997.
- [137] Arizona State University. Video Traces for Network Performance Evaluation. <http://trace.eas.asu.edu/>.
- [138] Rusty Russell. Linux 2.4 Network Address Translation (NAT) HOWTO. <http://www.iptables.org/>, January 2002.
- [139] S. Masoud Sadjadi, Philip K. McKinley, Betty H.C. Cheng, and R.E. Kurt Stirewalt. TRAP/J: Transparent generation of adaptable Java programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, pages 1243–1261, Agia Napa, Cyprus, October 2004.

- [140] S. Masoud Sadjadi, Philip K. McKinley, and Eric P. Kasten. Architecture and operation of an adaptable communication substrate. In *Proceedings of the Ninth IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, pages 46–55, San Juan, Puerto Rico, May 2003.
- [141] Carsten Schill. SMCRoute. <http://www.cschill.de/smcroute/>.
- [142] Margaret Pinson and Stephen Wolf. *Video Quality Measurement User's Manual*. U.S. Department of Commerce, February 2002.
- [143] Jun Xin, Chia-Wen Lin, and Ming-Ting Sun. Digital video transcoding. *Proceedings of the IEEE*, 93:84–97, January 2005.
- [144] Anthony Vetro, Jun Xin, and Huifang Sun. Error resilience video transcoding for wireless communications. *IEEE Wireless Communications*, 12:14–21, August 2005.
- [145] Ishfaq Ahmad, Xiaohui Wei, Yu Sun, and Ya-Qin Zhang. Video transcoding: an overview of various techniques and research issues. *IEEE Transactions on Multimedia*, 7:793–804, October 2005.
- [146] Javed I. Khan and Qiong Gu. Network aware video transcoding for symbiotic rate adaptation on interactive transport. In *IEEE International Symposium on Network Computing and Applications*, pages 201–212, Cambridge, MA, USA, October 2001.
- [147] Ian F. Akyildiz, WellJan Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40:102–114, August 2002.
- [148] Charles J. Turner and Jennifer G. Turner. Adaptive data parallel methods for ecosystem monitoring. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 281–290, New York, NY, 1994. ACM Press.
- [149] Bugra Gedik and Ling Liu. A scalable peer-to-peer architecture for distributed information monitoring applications. *IEEE Transactions on Computers*, 54(6):767–782, 2005.
- [150] Fabio Kon, Roy H. Campbell, and Klara Nahrstedt. Using dynamic configuration to manage a scalable multimedia distribution system. *Computer Communications*, 24(1):105–123, 2001.
- [151] Chuan Wu and Baochun Li. Echelon: Peer-to-peer network diagnosis with network coding. In *Proceedings of the Fourteenth IEEE International Workshop on Quality of Service (IWQoS)*, pages 20–29, New Haven, Connecticut, USA, June 2006.
- [152] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 49–60, Washington, DC, USA, 2006. IEEE Computer Society.
- [153] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE 2006)*, volume 4290 of *LNCS*, Melbourne, Australia, November 2006. Springer-Verlag.

- [154] Chiping Tang and Philip K. McKinley. On the cost-quality tradeoff in topology-aware overlay path probing. In *Proceedings of International Conference on Network Protocols (ICNP)*, pages 268–279, November 2003.
- [155] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.
- [156] Jin Xiao and Raouf Boutaba. QoS-aware service composition and adaptation in autonomic communication. *IEEE Journal on Selected Areas in Communications*, 23(12):2344–2360, December 2005.
- [157] Shansi Ren, Lei Guo, and Xiaodong Zhang. ASAP: an AS-aware peer-relay protocol for high quality VoIP. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 70–79, Lisboa, Portugal, July 2006. IEEE Computer Society.
- [158] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, pages 277–289, Asilomar, CA, USA, January 2005.
- [159] Jin Liang and Klara Nahrstedt. Service composition for advanced multimedia applications. In *Proceedings of 12th Annual Multimedia Computing and Networking (MMCN'05)*, volume 5680, pages 228–240, San Jose, California, January 2005.
- [160] Boris Jan Bonfils and Philippe Bonnet. Adaptive and decentralized operator placement for in-network query processing. *Telecommunication Systems*, 26(2-4):389–409, 2004.
- [161] Yanif Ahmad, Ugur Çetintemel, John Jannotti, Alexander Zgolinski, and Stanley B. Zdonik. Network awareness in Internet-scale stream processing. *IEEE Data Engineering Bulletin*, 28(1):63–69, 2005.
- [162] Sangeetha Seshadri, Vibhore Kumar, and Brian F. Cooper. Optimizing multiple queries in distributed data stream systems. In *Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW)*, pages 25–30, Atlanta, GA, USA, April 2006. IEEE Computer Society.
- [163] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceeding of the 28th International Conference on Software Engineering (ICSE '06)*, pages 371–380, Shanghai, China, 2006. Received ACM SIGSOFT distinguished paper award.
- [164] Ji Zhang, Zhinan Zhou, Betty H.C. Cheng, and Philip K. McKinley. Specifying real-time properties in autonomic systems. In *Innovations in Systems and Software Engineering*. in print.
- [165] Service-Oriented Network Architecture (SONA). <http://www.cisco.com/go/sona/>. Cisco.

- [166] Eric P. Kasten, Philip K. McKinley, and Stuart H. Gage. Automated ensemble extraction and analysis of acoustic data streams. In *Proceedings of the First International Workshop on Distributed Event Processing, Systems and Applications (DEPSA), in conjunction with ICDCS 2007*, Toronto, Ontario, Canada, June 2007.

MICHIGAN STATE UNIVERSITY LIBRARIES



3 1293 02956 4147