An Architecture for Availability and Performance in Wide-Area Service Composition

by

Bhaskaran Raman

B.Tech. (Indian Institute of Technology, Madras) 1997
M.S. (University of California at Berkeley) 1999

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Professor Ion Stoica
Professor David R. Brillinger

Fall 2002

The dissertation of Bhaskaran Raman is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2002

An Architecture for Availability and Performance in Wide-Area Service Composition

Abstract

An Architecture for Availability and Performance in Wide-Area Service Composition

by

Bhaskaran Raman

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Randy H. Katz, Chair

With the growth of wireless telecommunications and the mobile Internet, there is an increasing demand for mobile data services. Novel value-added services and content provisioning are expected to be the driving force behind the deployment of future communication networks and the mobile Internet. In this context, we consider the idea of *service composition*, where independent service components deployed and managed by providers are assembled to enable a novel composed service. A simple example of service composition is a scenario where a video-on-demand service is enabled on a thin client by composing it with a video transcoder service component. Composition achieves flexible service creation through the reuse of the individual components. We term the set of service components in a composition, along with the network path in-between, as a *service-level path*.

In this thesis, we address the issues of *availability* and *performance* in such service composition. When a service-level path spans multiple Internet domains (the wide-area Internet), its availability is affected by that of the underlying Internet path. Studies have shown that inter-domain Internet path availability is poor, and when inter-domain path failures happen, Internet route recovery can take several tens of seconds to several minutes. We address this issue by (a) *detection* of failures on the Internet path that constitutes a leg of the service-level path, and (b)

recovery, by choosing alternate network paths and/or alternate service replicas. For performance of the service-level path, it is important to choose appropriate service instances to be composed, and ensure load-balancing among the service replicas.

We address several challenges with respect to availability as well as performance of composed services. We first study the issue of failure detection by analyzing long traces collected over several wide-area Internet paths. We find that it is possible to define a notion of failure, lasting for long periods of time such as several tens of seconds, quite distinct from intermittent congestion. More interestingly, we find that such long outages can be predicted with timeouts of much shorter durations – within about 1.2-1.8 seconds on most Internet paths. Further, we find that such timeouts happen quite infrequently in absolute terms, only about once an hour or less. These observations point at the usefulness and feasibility of a failure detection mechanism.

As a next step to the study of failure detection, we develop an architecture for wide-area service composition based on an overlay network of service clusters. The use of clusters as a unit of construction allows us to separate the issue of process- or machine-level failures of service components, from the issue of network-level failures. The overlay network allows us to define alternate service-level paths, as well as balance load across the various cluster execution platforms. An important feature of the overlay network is that it is a *virtual-circuit* based network, as opposed to being a datagram-based network. This has the important implication that we can do service-level path failure recovery without waiting for the failure information to propagate and stabilize across the network.

Using an emulation testbed, we find that the scaling bottleneck is the per-session switching state at the overlay nodes. However, an off-the-shelf Pentium-III 500 MHz machine can easily handle the per-session switching state of at least 250 simultaneous client sessions. This amounts to little additional provisioning in comparison to the provisioning required for the actual service components.

We perform an in-depth study of the issue of performance of service-level paths from the point of view of load-balancing among server replicas. We introduce the *least-inverse-available-capacity (LIAC)* metric for choosing a set of service instances for a composed client session, as well a *piggybacking* mechanism to update load information via the service-level path setup messages. The quick feedback provided by the piggybacking mechanism is effective in achieving load-balancing in time as well as space (across the different service replicas).

We illustrate the use of our architecture by presenting a set of application scenarios involving service composition. These scenarios are in the context of the *Universal Inbox* architecture that achieves extensible any-to-any communication in a heterogeneous network. We choose a specific implementation of an application, a composed text-to-speech service, to study the usefulness of our architecture. We deploy the composed application as well as our recovery mechanisms in a wide-area testbed and run long-running experiments to evaluate the improvement in availability due to our recovery algorithms. We observe dramatic improvements in availability. In quantitative terms, we are able to reduce the system downtime by factors of up to 2-10 in many cases. This shows the usefulness of our architecture from the point of view of the end-user.

Professor Randy H. Katz
Dissertation Committee Chair

To my parents,

my mentor, Dilip,

and my friends at Berkeley.

Contents

List of Tables

List of Figures

Acknowledgements

"It is the process that is more important than the result", says my adviser. I owe both the process of my PhD, my overall development as a researcher, as well as the end result, this thesis, to my adviser, Professor Randy H. Katz. His constant guidance and mentoring for the last five years has been a great source of support for me. I have always wondered at how he is able to provide incisive and immensely useful feedback on my writings, presentations, as well as my thought process in research. My one-on-one meetings with him have always made me feel infinitely more confident and energetic. Despite his very busy schedule, he always makes time for his students; never once has he made me feel like I need more of his time than he is giving me. His time management, discipline, mentoring skills, teaching as well as researching expertise are things I have learned a lot from, and would seek to emulate in my own way as I pursue my career. Randy has been the very best adviser I could have hoped for.

Apart from my adviser, throughout the duration of the ICEBERG research project, I have interacted with, and learned from Prof. Anthony Joseph. From providing guidance on testbed implementations to shaping of research ideas in brainstorming sessions, Anthony too has been a great source of support for me. Anthony was one of the members in my qualifying examination committee, and I thank him for his feedback during that phase of development of this thesis. Prof. Ion Stoica, and Prof. David Brillinger also were part of my qualifying examination committee, as well as my dissertation committee. I thank them for their time, support, and feedback on the thesis at various stages.

My first research project at Berkeley was in collaboration with Vijayshankar Raman, under Prof. Joseph Hellerstein. I really enjoyed working with Vijayshankar and Joe and learned from the experience. During my first couple of years at Berkeley, I also learned a lot from research projects related to various courses taught by Prof. Steve McCanne, Prof. Anthony Joseph, Prof. David Culler, and Prof. Eric Brewer. As much as I enjoy research, I enjoy teaching as well. I thank Prof. Brian Harvey, Prof. Richard Fateman, Prof. David Forsyth, as well as all my fellow teaching assistants for making my role as a teaching assistant during my first two semesters a pleasant and enjoyable experience.

Through the years, I have come to appreciate the excellent, beyond-compare academic environment Berkeley has to offer. The greatest thing about Berkeley is its collaborative atmosphere. I have learned as much from my fellow students as from the professors. While, some have played the role of a mentor at times, as I progressed through my PhD, some have played the role of a mentee. From brainstorming sessions on research ideas to heated debates, from helping with system administration to debugging code, from feedback on written material to critique on presentations, my fellow students at Berkeley have been a constant source of support. This of course, apart from their invaluable friendship and company.

Hari Balakrishnan, Venkat Padmanabhan, Mark Stemm, Todd Hodes, Yatin Chawathe, Steve Gribble, Armando Fox, David Wagner, Ian Goldberg, and Tina Wong were my "senior" graduate students, and always a great source of inspiration. In the ICEBERG research group, I have enjoyed working with Sreedhar Mukkamalla, Jimmy Shih, Ben Zhao, Steve Czerwinski, Kim Keeton, Barbara Hohlt, Almudena Konrad, Amoolya Singh, Helen Wang, Morley Mao, and Chen-Nee Chuah. From the twin project NINJA's research group, I enjoyed interacting with Mike Chen, Nikita Borisov, Matt Welsh, and Rob von Behren. As the ICEBERG research group transformed itself into the SAHARA project, I worked with Sharad Agarwal, Mukund Seshadri, Sridhar Machiraju, Lakshminarayanan Subramanian, Matthew Caesar, Weidong Cui, Kevin Lai, and Yan Chen. I'm grateful to all of them for being such wonderful fellow graduate students. I also thank Kirby Zhang, Adam Costello, Drew Roselli, Giao Nguyen, Tom Henderson, Angela Schuett, Ramakrishna Gummadi, Andrew Begel, as well as all my fellow "birk-ball" players for their friendship and company.

I thank Kathryn Crabtree, Peggy Lau, La-Shana Polaris, Bob Miller, Nathan Berneman, Glenda Smith, Terry-Lessard Smith, and Damon Hinson for handling all the administrative details throughout my years at Berkeley. Keith Sklower, Brian Shiratsuki, Eric Fraser, and Albert Goto have helped me a lot with system administration and testbed implementations for my research projects. I'm grateful to them for this support. I must also thank the various sponsors of my research projects, specifically Ericsson. I have enjoyed working with the various researchers from

Ericsson: Reiner Ludwig, Stephan Baucke, and Per Johansson, who have spent varying amounts of time at Berkeley.

I interned at IBM, T. J. Watson research center during the summer of 1999. I really enjoyed my interaction with the researchers at IBM. I especially thank Dilip Kandlur, my manager, and my mentors Pravin Bhagwat, and Srinivasan Seshan. During the three months at IBM, I was fortunate to have the friendship and company of Kamakshi Sivaramakrishnan.

Like many others who study at Berkeley, I have come to the conclusion that Berkeley is the best place on earth. As much for its extra-academic environment, as for its academics. I have been fortunate enough to meet some of the most amazing, warm, and loving people at Berkeley. I thank Ashwin Nayak, Yatin Chawathe and Sundar Kumar Iyer (SKI) for helping me during my first few days at Berkeley. SKI has been one of the greatest sources of inspiration in my life. The warmth he exudes can be felt almost immediately and fills me with such energy and makes me aspire to be a fraction as humble as him. SKI also introduced me to the wonderful organization, "Asha for Education". Asha strives to promote basic education for underprivileged children in India. I have been part of this organization for the last four years, and have developed close friendships with several other great people.

Arvind Krishnamurthy, Richa Govil, and Shailen Mistry were among the first people I met as fellow volunteers in Asha. Another source of inspiration for me through Asha is D. P. Prakash. Although I have not interacted with him much in person, I could feel his energy field all the way across the country from Vermont where he lives! Through the years in Asha, I interacted with Shashi, Prashanth, Praveen, Gopal, Ganesh, Neha, Anand, Amritha, Sriram, Akanksha, Nirmal, Charu, Vishy, Kameswari, Champaka, Renu, Barnali, Savitha, Ashwin, Manali, Rahul, Shruti, Akhila, and several others. I consider my set of friends at Berkeley to be the most valuable treasure I have earned during my stay at Berkeley. Shashi, Prashanth, Anand, Amritha, Charu, Vishy, Neha, and Gopal have been my greatest source of support during some of the toughest times I faced.

Dilip is my mentor on philosophy and life. My interaction with him started during my undergraduate years at IIT-Madras. He is the one who has made me find my inner energy, the reason for doing what I do, and the purpose of my life.

Last but not the least, my parents have shown such immense support and patience throughout my studies. It would be a mistake for me to try and express their unconditional love for me in words.

I dedicate this thesis to my parents, my mentor, Dilip, and all my friends at Berkeley.

Chapter 1

Introduction and Motivation

The past decade has seen tremendous growth in communication technology. There has been parallel evolution in mobile telecommunications as well as the Internet. Cellular telecommunications systems have evolved from analog-based first generation systems in the 1980s to digital second generation systems. The most widely deployed second generation system today is the Global System for Mobility (GSM). GSM has exhibited growth from early stages in 1991 to over 250 million subscribers in 2000 [59].

The Internet has shown exponential growth as well, with the amount of data traffic on the Internet exceeding all voice traffic in the year 2000 [59]. There is a growing trend toward convergence of mobile telecommunications systems with the Internet, and increasing demand for data services on cellular systems. In Japan, i-Mode has shown the tremendous potential for data services and mobile Internet [5]. Next-generation telecommunications systems such as UMTS (Universal Mobile Telecommunications System) [68] and IMT-2000 (International Mobile Telecommunications) [65] aim to provide high-speed data through their access networks.

With growing demand for mobile data services, novel value-added services and content provisioning will be the driving force behind the development and deployment of future communication networks and the mobile Internet. The value chain is expected to undergo evolution as the dominant part of the revenue moves from the network operator to the service and content providers.

Next generation network architectures such as UMTS admit a service architecture in which service delivery is an integral part of the value chain [59]. Figure 1.1 from [59] depicts the service delivery model in UMTS. The term *service* here means end-user application functionalities such as content access, personal information access, or other communication functionalities.

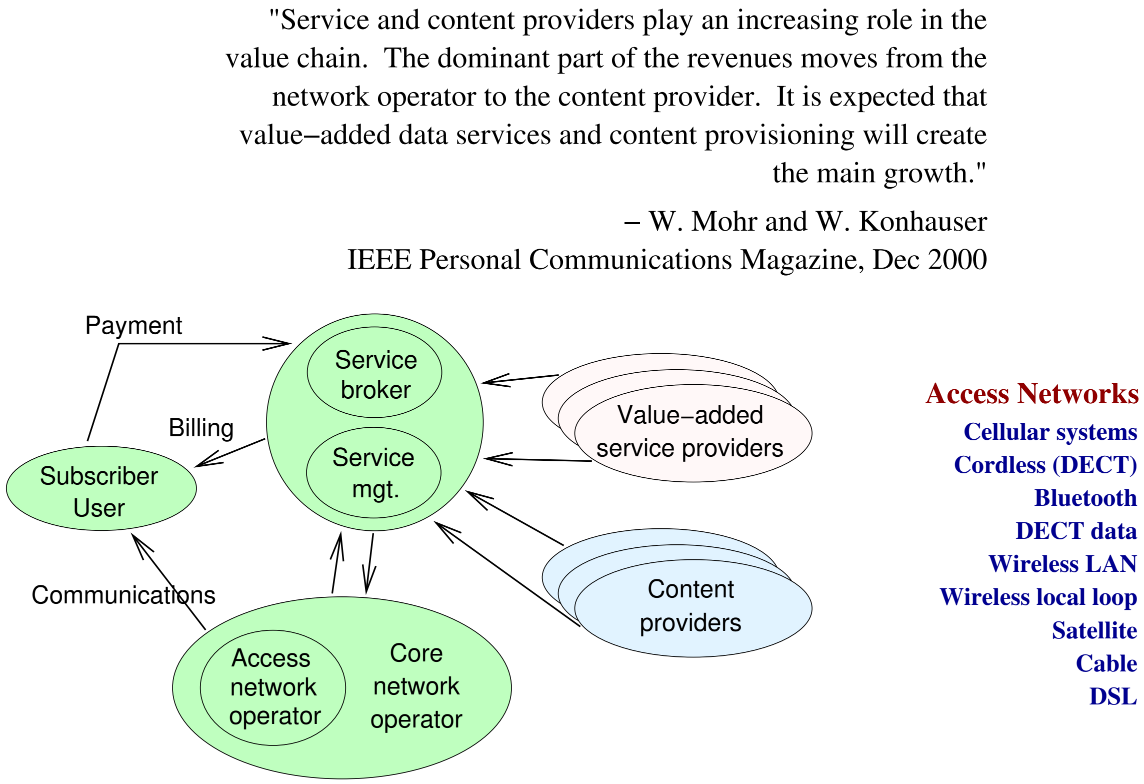

Figure 1.1: Importance of value-added services in future communication networks [59]

The open service model of the Internet has been responsible for the rapid development of a wide variety of web-based services [46]. As the Internet evolves and expands to include next-generation, data-enabled mobile communication networks, it is important to make data services available on these networks rapidly. It is essential to enable quick and flexible development and deployment of end application functionality. Such functionality should be enabled in the presence of a wide variety of heterogeneous access networks. *Service mobility* is the feature where the user can access the same set of services seamlessly, independent of the access network and the access device. To achieve this, it is necessary to personalize and adapt content to end-user devices with varying capabilities. In this context, we consider the idea of *composition*.

1.1 Service Composition

Composition in a generic sense refers to the construction of a larger functionality by assembling smaller, simpler components. Composition is a well studied idea in the domain of software design [72, 73]. A simple and well-known example of composition technique is that of Unix piping. An example is shown in Figure 1.2. Independent software components (programs) are assembled to enable a larger end-user functionality.

Figure 1.2: Unix piping: an example of composition

Composition allows easy and flexible development of new functionality since the individual components are reused for multiple compositions. New functionalities that were never thought of during the design of the individual components can be enabled rapidly.

In this work, we consider composition of *service components* that are deployed and managed by *service providers* at different points on the Internet. We illustrate this with a couple of examples. Figure 1.3 depicts a scenario where a user has a novel thin client device. The device is network enabled with a wireless access technology that has bandwidth limitations in the access network. The user roams to a foreign network and wishes to access a local news/weather video service. A portal provider enables this by composing the video service with an appropriate transcoder to adapt the contents of the video to the thin client's capabilities.

Next, she wishes to access her email from her home provider on her cell-phone while she is on the move. The portal provider enables this by composing a text-to-speech conversion engine, with the user's home email repository.

In both these examples, a novel application functionality is achieved on the Internet-enabled access device/network. Service mobility is achieved by making content (video or email) available in a seamless fashion, in a network-independent and data-type-independent manner. In either case, the components could potentially belong to different service providers, as indicated in the figure. The portal provider who does the composition, and interacts with the user, may be a third-party. Such composition can reuse the individual components not only in implementation, but also in deployment. The same deployed and managed instance can be used for multiple compositions. This offers a flexible way to enable new application functionalities.

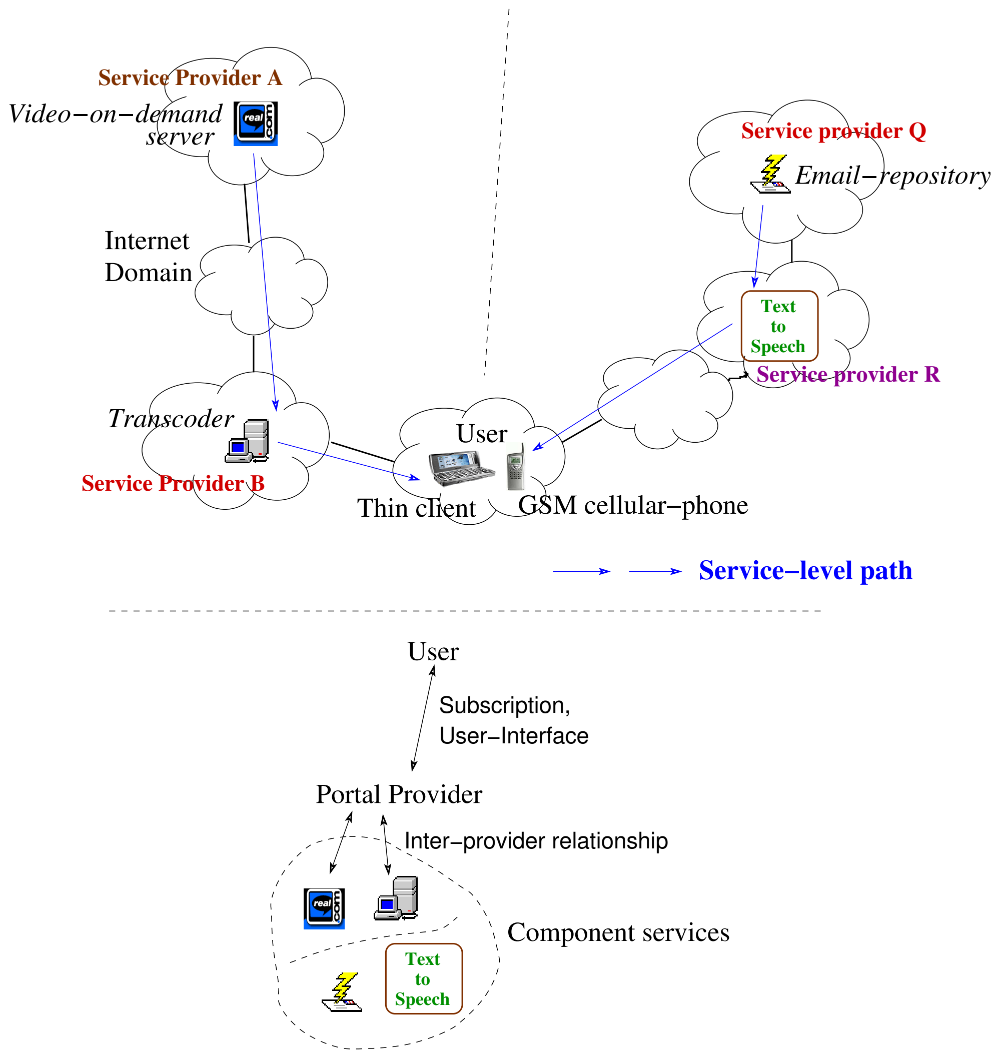

Figure 1.3: Composition of independent service components across the network

We use the term "service" to refer to the components in composition, as well as to refer to the composed application functionality. We qualify the term where necessary. In a composition, we term the set of component services along with the network path in-between as a *service-level path.* Since our focus is on service composition, we sometimes use the unqualified term "path" to refer to a service-level path.

We envision a wide variety of composable multi-media services and middle-ware components such as media transcoding agents (audio/video), protocol transformation components, rate-adapting agents, media transformation engines, redirection or filtering agents, personalization, customization or user-interface agents, etc. We have implemented several such components as part of the ICEBERG project [67, 81], to achieve service integration across heterogeneous networks. The IETF Open Pluggable Edge Services (OPES) group also talks about such composable components in the context of web-services [27]. Table 1.1 gives a few concrete examples of such components. These would be deployed and managed by a variety of service providers and be available for composition of new applications for novel devices in next-generation networks.

Composable service component	**Example(s)**
Media transcoding agent	MPEG video $\leftrightarrow$ H.261 video, H.729 audio $\leftrightarrow$ PCM audio
Protocol transformation agent	Multicast to unicast proxy server
Rate adaptation agent	Reducing video frame rate, Removal of video color, to reduce bandwidth requirement
Media transformation agent	Text $\leftrightarrow$ PCM audio, (Speech synthesis, Voice recognition engine)
Redirection agent	Proxy to redirect video to user's desktop or handheld based on user location
Filtering agent	User proxy to filter spam
Personalization/user-interface agent	Proxy to adapt user-interface to user-device
Addition of semantic content	Adding news/stock-quote/ads to video stream, Proxy to add song title in audio stream

Table 1.1: Examples of service components for composition

1.2 Challenges and Issues in Wide-Area Service Composition

Composition of software components can be of three types depending on the components' interfaces: using *program interfaces*, *user interfaces*, or *data interfaces* [73]. Program-interface based composition uses programming language abstractions such as procedures or class hierarchies for software reuse. An example of user-interface based composition is the case of pseudo-ttys, or X-servers in UNIX, to reuse programs. The same program can be run on a local terminal, a dumb terminal, or from a remote machine with an X-server.

Data interfaces involve data being passed on from one component to another. UNIX pipes and filters fall under this category. Our examples of service composition fall under this category too. In our context, we consider composition of software components that are deployed and managed by multiple service providers at different points on the Internet. The data interface based composition thus translates to a data flow across the network. This brings forth several critical issues and challenges. We discuss these issues under two broad categories: (1) *availability* of the composed service, and (2) its *performance*.

Issues related to availability

When providers deploy service instances independently, the composed service-level path could span multiple Internet domains. This is shown in Fig. 1.3. This has implications on the *availability* of the composed service.

Service providers would like to have their services always available for client sessions. To give an idea of the desirable availability standards, the Public Switched Telephone Network (PSTN) achieves five 9's availability [52]. That is, 99.999% availability. This translates to about 5 minutes of downtime per year. However, studies have shown that inter-domain Internet path availability is very poor [55]. Availability can be as low as 95% in some cases. More importantly, when inter-domain path failures happen, Internet route recovery can take several tens of seconds to a few minutes [54]. This in turn reflects on the availability of the composed service. An inter-domain path failure is depicted in Figure 1.4.

The figure shows an Internet path failure in the middle of a composed client session. This points at the following issue. Availability of long-lasting sessions is affected by failures in two different ways. Reachability between service components along the service-level path is required during (a) session setup, as well as (b) during the session. Multimedia sessions could last for several minutes to hours. In comparison, session setup is likely to take a much shorter duration (a few hundred milliseconds to may be a few small number of seconds). In such a scenario, it is more important to address network failures *during* a session, to improve the overall availability.

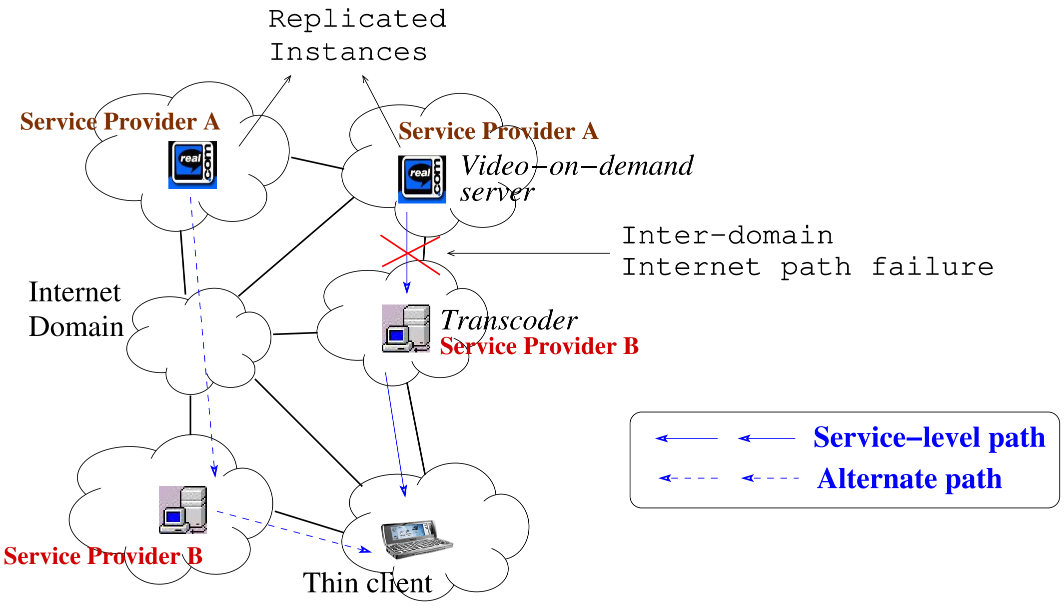

Figure 1.4: Inter-domain failures, and use of alternate service-level paths

Our approach to address this issue involves: (1) monitoring the service-level path; that is, monitoring the network path between successive components in the composition, for failures, and (2) using alternate service replicas as well as alternate Internet paths when the original service-level path experiences an outage.

In taking such an approach, we do not depend on the underlying network for either failure detection or for recovery. The dotted lines in Fig. 1.4 show one such alternate service-level path for the composed video session.

There are two main steps in this approach:

1. *Detection* of failure on an Internet path that constitutes a leg of the service-level path, and

2. *Recovery*, after failure detection, by choosing an alternate network path and/or alternate service replicas.

There are issues and challenges to be addressed in both these steps. The issues with respect to failure detection are as follows.

1. As in Figure 1.4, the network between successive components in a composition could be an inter-domain path. There is inherent variability in delay, loss-rates, and outage durations on an inter-domain Internet path. We have used the term "failure" loosely so far. It is an open question as to whether there is a clearly defined notion of failure. Intuitively, a failure is a long-lasting outage in the network, when no packets go through from one end of the Internet path to the other. We are especially concerned with BGP-level failures where recovery takes several tens of seconds to several minutes [54]. Intermittent congestion could last for varying periods of time (few hundred milliseconds to several seconds). Short-term congestion losses may be indistinguishable from failures. There could also be intermittent short outages due to a variety of factors (such as intra-domain route change, router reboot, etc.). In such a case, a clear definition of failure becomes difficult. Reacting to all congestion loss through a recovery mechanism will only add to the system overhead.

2. Second, if there exists a clear notion of a failure, it is also an open question as to how quickly these failures can be detected. It is important to ensure that failure detection and recovery is *quick* for real-time applications. Ideally, recovery within a few hundred milliseconds is suited for interactive applications. Recovery even within the limits of a few small number of seconds (3-4 seconds) would be very useful for applications. This is especially so for non-interactive applications such as streaming of stored video, which usually have buffers of 5-10 seconds. Detection and recovery within these bounds can potentially hide long-lasting inter-domain outages from the end user, thus improving the overall availability.

   "Quick" failure detection is especially interesting given that there is no support for such detection from the underlying Internet. Since we are considering inter-domain paths that span multiple ASes (Autonomous Systems), such support is especially difficult to achieve. Commercial ASes do not like to share or reveal information about their internal network structure, protocols used, outage patterns, etc.

3. The third open question with respect to failure detection is, given a notion of failure, and a detection mechanism, how often is intermittent congestion confused for a failure. Intuitively, an aggressive failure detection mechanism with a small timeout may trigger *spurious* path restorations, where we confuse intermittent congestion in the Internet path with a failure. On the other hand, a conservative failure detection mechanism with a large timeout could mean longer detection times in general.

In short, the question to answer is: *how quickly (within a few small number of seconds) failure detection can be done with minimal overhead in terms of spurious failure detection timeouts.*

The process of recovery, after failure detection, poses challenges as well. *Scalability* is an important consideration. The system can scale in multiple dimensions:

- *Number of clients:* as the system grows, the number of clients using composed services can be expected to increase.

- *Expanse of the system:* ideally we would like the system to be able to operate on an Internet-wide distributed scale. Note that this dimension is independent of the number of clients – we could have a small number of clients dispersed all over the Internet.

- *Number of service instances:* with increasing number of clients and larger expanse, we can expect the system to have multiple replica instances for each kind of service.

Recovery is especially challenging when the system scales in the number of clients. Recall that sessions can be long-lasting, and hence the number of *simultaneous* client sessions grows with the number of clients. This means that a large number of client sessions may have to be restored simultaneously when a failure is detected.

Failure detection and recovery are challenging when the system scales to a large expanse and has a large number of service replicas. Collecting and maintaining current network reachability and performance information between service component instances becomes harder.

Issues related to performance

Apart from the issue of availability, there is also the issue of *performance* of the composed session. For a particular client request, we have to choose the right set of service instances to instantiate the session. This is challenging due to at least two reasons:

- Unlike traditional web mirror selection, we have to choose a *set* of service instances for each client session, and not just one web mirror replica. We have to choose a service-level path with lightly loaded service instances and adequate performance along the network path. Balancing load among the various service replicas is essential.
- We not only have to choose service instances during service-level path creation, but also during recovery. This adds an additional dimension to the dynamics.

These issues become harder as the system scales in the three dimensions mentioned above. Load balancing in an Internet-wide system is an interesting issue. Load balancing in any distributed system consists of several components including: (a) a feedback loop between the point where load is experienced and the point where decisions are made, and (b) a mechanism to use the feedback to drive future decisions of where to place load. These have to be designed to prevent load oscillations and to provide stable behavior under a variety of conditions. This becomes challenging when the load due to a large number of clients is involved, and when the feedback loop necessarily has to include the wide-area latencies.

In this work, we seek to address these issues and challenges in wide-area service composition.

1.3 Goals and Assumptions

In light of the issues presented in the previous section, our goals in wide-area service composition are:

- **Availability:** We wish to detect and recover from failures quickly, so that real-time composed sessions see minimal or no interruption. We especially focus on failures in inter-domain network paths, which can be long-lasting (several 10s of seconds), and where inter-domain coordination is difficult due to operational reasons. Avoiding long-lasting outages would improve the overall availability of the system. We wish to push system availability as close to 100% as possible.

- **Performance:** We need to choose appropriate lightly-loaded service instances and the appropriate network paths during service-level path creation as well as recovery. We wish to balance load across the various service replicas. The interplay between recovery and the choice of lightly loaded service instances is important too.

- **Scalability:** Both the goals of availability and performance should be achieved while considering system scalability under different dimensions: number of clients, expanse of the system, and number of service instances.

In our work, we make some assumptions about the operational model with respect to service composition. We envision an scenario where independent service providers deploy and manage their instances at different points on the Internet. Such a scenario is likely to emerge for deployment of novel services in 3G+ networks. We assume that services are run and managed by their providers, and they are not light-weight mobile code. That is, a service instance cannot be moved from one network location to another for the purposes of a particular composed client session. This is a very reasonable assumption and reflects the way services are deployed on the Internet today. Providers like to have control over their service instance code and its performance, and do not like to have their code executing on platforms outside their control.

In our operational model, we also assume that the composition is performed by a portal provider. The portal provider chooses which services (not the specific instances, but just the kind of services) are composed for a particular application functionality. This portal provider interacts with the component service providers as well as the end user. (See Fig 1.3). This assumption allows us to separate the issue of service interfacing in composition. The portal provider is responsible for ensuring that the data interfaces between the components in composition match one another. A wide variety of compositions are possible even with this assumption in place, as illustrated in the examples earlier. And this is likely to be the way services are composed before automatic service interface description and service interface matching issues are addressed.

In our mechanisms for service-session recovery, we assume that the services in the service-level path have only *soft-state*, and no persistent state. Soft-state is state that can be built up by an alternate instance of the service without affecting *correctness* of the end-to-end composed service. There may be a performance issue with respect to rebuilding soft-state at an alternate instance, but no issue of correctness. Our mechanisms do not attempt to synchronize, or in any way maintain persistent application level service state. (For a discussion of persistent service-state, see [44]). Distributed state consistency in a loosely coupled system like the Internet is a difficult problem, since network partitions are unavoidable on the Internet [41]. This assumption about soft application state is however, not particularly restrictive: all of the services we listed earlier – such as content adaptation, content addition/transformation, etc. fall under this category.

1.4 State of the Art

The concept of composition and its different forms using the different kinds of interfaces (program, user, and data interfaces) are well known. Object oriented systems have been using software composition with class hierarchies for code reuse. Composition of software components that are of larger granularity such as kernel modules, or UNIX programs, are well known.

We are considering composition at an even coarser granularity: independent service components deployed and managed by potentially different service providers. Our goal is to address the networking and distributed systems issues in the design of an architecture for such composition. We now briefly summarize past work in related dimensions.

1.4.1 Wide-area composition

With growing software complexity and development of distributed client-server systems, object-based composition using software components across the network was a natural outcome. Common Object Request Broker Architecture (CORBA) is an architecture, developed starting in 1989, for composition of objects in a distributed system [61]. It allows remote procedure calls (RPCs) for client-server communication in a platform and operating system independent manner. Distributed Component Object Model (DCOM) is Microsoft's commercial implementation of an object-based system [4], and is CORBA's main competitor. With the development of Java, its Remote Method Invocation (Java-RMI) provided a mechanism for composition of Java objects across the network.

Such composition falls under the category of program-interface based composition. Here the focus is on the definition of interfaces, specification of network-byte-level coding for objects, and on programmatic handling of various erroneous conditions that might arise in a distributed system. Long-running client sessions involving real-time data are not the focus in this scenario, and the issues of availability and performance have not been considered in this domain. An essential difference between object-based systems and our wide-area service composition is that in the former, object state consistency is a more important consideration and comes at the cost of system availability. Whereas, since we do not worry about state consistency (service components have only soft-state), we achieve greater system availability.

The web-services initiatives [80] extend object-based distributed composition to generic services on the Internet, and also consider data-flow based composition. WSDL (Web Services Description Language) [37] is an XML-based language to define services. And WSFL (Web Services Flow Language) [56] describes a composition by defining the "flow" of control and data in the composition. The Open Grid Services Architecture (OGSA) [11] adapts web-services for scientific grid computing by defining functional components for service discovery (the Meta Directory Service), dynamic service creation, lifetime management, and security (the Grid Security Infrastructure). The service description and interface languages of the web-services domain, as well as the service discovery, security, and management functionalities enabled in OGSA are complementary to our work. They address issues orthogonal to the availability and performance aspects that we focus on in this thesis.

The IETF OPES (Open Pluggable Edge Services) group [12] defines a framework for services that can be "plugged", or composed [25]. The application scenario is similar to ours in that it involves a data-flow based composition of intermediary service components. However, the OPES framework does not include mechanisms for recovery when a composed session fails. Nor does it deal with issues of distributed load balancing in the context of composition. The OPES effort includes the Intermediary Rule Markup Language (IRML) [26] to enable the content source to specify conditions under which (parts of) its content may be transformed or adapted for a particular client. Such a rule specification language is complementary to our work on availability and performance issues.

ALAN (Application Layer Active Network) [42] proposes application-layer routing by proxylets. Such application-layer routing is similar to data-interface based composition across the network, where the data passes through multiple service components (the proxylets). However, the operational model in ALAN is different in that the proxylets can be dynamically created and moved around. In our case, the services are deployed by different service providers, and are heavy-weight in nature. Also, ALAN does not have quick-recovery from failures as one of its goals. In our work, we specifically evaluate the recovery aspect of the system in the context of real-time composed services.

1.4.2 Cluster-based availability

Availability is an important feature for commercial services. This issue has received wide attention in the context of web servers. Cluster-based platforms have evolved as an effective mechanism to scale web-servers as well as to achieve resilience to server failures. Locality-Aware Request Distribution (LARD) [63] explores mechanisms for intelligent redirection of incoming client requests among a cluster of servers. Commercial level-4 switches are available to choose between server replicas within a cluster [3].

With the evolution of the web and growing heterogeneity in Internet hosts and access networks, proxies evolved a mechanism for adaptation of content across this heterogeneity. The use of proxies is a form of composition at a coarse granularity, since the proxies are rather independent of the servers or the clients.

The TACC (Transformation, Aggregation, Caching, and Customization) project [40] developed mechanisms for making content adaptation proxy components highly available. The TACC project also considered composition of these components. However, an important difference is that TACC considered composition of components within a single cluster of workstations, where all the different components are close together. Thus, wide-area network path failures, their detection, and recovery were not factors. This is true of cluster-based web-servers as well – while they consider server failures, they do not address wide-area network path failures. TACC developed mechanisms for handling process/machine level failures in service components, and this is complementary to our mechanisms for handling network path failures.

The Active Services model [19] provides a cluster-based mechanism for handling process/machine level failures as well. It uses multicast based soft-state heart-beats from the client to achieve this. Again, network failures between components in a composition, quick detection and recovery, are not considered here.

1.4.3 Load-balancing across the wide-area

Load-balancing and availability issues across wide-area service replicas have received a great deal of attention in the domain of web-servers. Several mechanisms have been proposed for load balancing of distributed web-server systems [33]. These include client-based approaches [83, 23, 76], DNS-based approaches [53, 77], dispatcher-based approaches [45], as well as server-based approaches [21, 29]. Service composition involves at least two novel aspects that pose new challenges. First, unlike web-mirror selection, we have to choose a *set* of service instances for each client. Second, we consider failure detection and recovery of composed service in the middle of a long-lived session. These lead to a consideration of a fundamentally different architecture that consists of an overlay network of service clusters over which services are composed.

1.4.4 Improving network availability through overlay networks

Routing around failures (above the IP level) in the wide-area has been addressed in other contexts. The RON project [20] uses an overlay topology to route around temporary failures at the IP level. In the specific context of video delivery, packet-path diversity has been used as a mechanism to get around failures in [22]. Content-addressable networks [69, 86, 79] also provide an overlay topology for locating and routing toward named objects. However, these mechanisms are not applicable for composed services – with composed services there is the constraint that the alternate recovery path has to include the component services as well. That is, we require not a simple network path, but a service-level path with service instances in-between.

1.4.5 Summary

In summary, past work that has considered composition has not addressed the wide-area availability and performance issues. And work that has considered availability and performance issues of servers or network paths has not considered service composition. The issues of availability, performance, and scalability have not been addressed in the context of composition of services across the wide-area Internet.

1.5 Thesis Methodology and Contributions

In this thesis, we develop and evaluate an architecture for wide-area service composition, addressing issues of availability, performance, and scale. Our system-design methodology is depicted in Figure 1.5. We begin with an initial *analysis* of the behavior of Internet paths (phase-I), to answer the crucial questions with respect to the notion of failures and failure detection. Based on this, we *design* an architecture for wide-area service composition (phase-II). The architecture relies on a service-level overlay network of cluster platforms. We address issues of load-balancing in the context of our architecture. We then *implement* a composed application (phase-III) and evaluate the usefulness of our failure recovery mechanisms from the point of view of the application (reiterate phase-I).

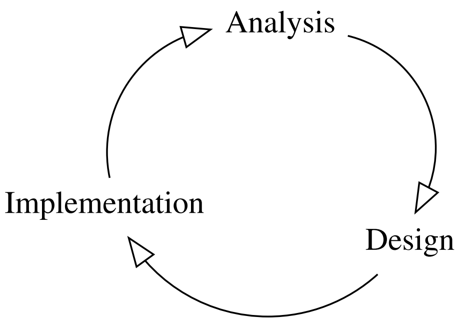

Figure 1.5: Thesis methodology

The methodology we adopt for the initial analysis is one of trace collection over wide-area Internet paths. The choice of a test platform for the design study (phase-II) is crucial. A simulation platform is inappropriate in this setting since simulation testbeds such as ns-2 [14] do not scale well with the number of clients. Also, simulations do not capture true processing and network bottlenecks. On the other hand, a wide-area testbed that spans the Internet is hard to setup and maintain. This also makes a controlled design study of the algorithms difficult, due to the non-repeatability of experimental conditions. Hence, for the design study, we develop an emulation platform. Here, we have an actual implementation of the various algorithms and mechanisms. The distributed components of our architecture run on different machines within the same high-speed LAN. The wide-area network characteristics such as latency and loss patterns are emulated. This allows us to perform a controlled design study to identify scaling bottlenecks and evaluate the various algorithms and design choices.

In phase-III, we carry over the implementation of our architecture from our design study, and implement a composed application on top of it. This allows us to reiterate phase-I through an evaluation from the point of view of the composed application. For this, we use a different testbed. While the emulation platform allows a design study, it does not show the usefulness of our architecture in terms of its goals of availability improvements. We use a real wide-area testbed consisting of machine spread across the Internet. This platform is smaller than our emulation platform in the number of nodes, but allows us to test our recovery algorithms on the Internet.

We now describe our thesis contributions in each of the phases of analysis, design, implementation/evaluation.

1.5.1 Failure detection on Internet paths

In designing a mechanism to keep track of the "liveness" of an Internet path, we do not rely on any network support to detect failures. We use an end-to-end heart-beat based control channel for this. To address the questions with respect to failures and their detection, we start with a study of Internet paths . We collect long-running traces of packets across a wide variety of Internet paths. These packets represent keep-alive heart-beats to monitor the liveness of the underlying Internet path. We study this to analyze if there is a clearly defined notion of a failure. We find the following:

- It is indeed possible to define a notion of failure, where no packets go through from one end of the Internet path to another for long periods of time such as several tens of seconds. This is quite distinct from intermittent congestion.
- More interestingly, we find that such long outages can be predicted with timeouts of much shorter durations (as compared to the failure durations), within about 1.2-1.8 seconds on most Internet paths. With such a definition of a failure, and this short a failure detection timeout based on heart-beats, we find that spurious timeouts happen quite rarely. Spurious timeouts are instances when we confuse an intermittent congestion or a short outage (lasting longer than the timeout period, but shorter than our definition of a long outage) with a long outage. Such spurious timeouts happen infrequently in absolute terms, about once an hour.

Intuitively, the small failure detection timeout indicates the *usefulness* of a detection and recovery mechanism for real-time applications, and the infrequent spurious timeouts point at the *feasibility* of employing a recovery mechanism. Encouraged by these, we proceed to design the recovery mechanism to build on top of the failure detection.

1.5.2 Architecture for wide-area service composition

As a next step to the study of failure detection, we develop a three layer model for wide-area service composition (Figure 1.6. At the lowest layer, we have cluster execution platforms that form the basis of our architecture. Next, these clusters form an overlay network among themselves. Service composition is performed on top of the overlay network of service clusters.

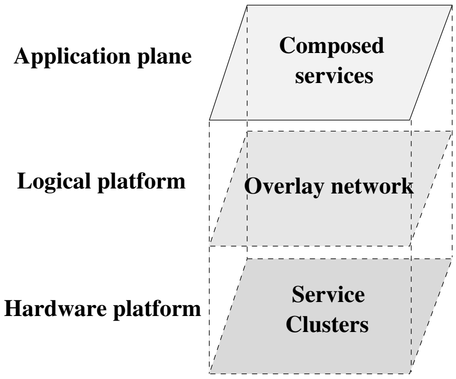

Figure 1.6: Architecture: Three layers

The architecture has several crucial features that allow us to achieve our goals. The use of clusters as a unit of construction allows us to separate the issue of process- or machine-level failures of service components, from the issue of network-level failures. It also allows us to scale in the dimension of the number of clients and number of service instances independently of scaling in the dimension of the expanse of the system across the Internet. The overlay network provides the context for exchange of network and cluster performance information. This allows us to define alternate service-level paths, as well as balance load across the various cluster execution platforms.

An important feature of the overlay network is that it is a *virtual-circuit* based network, as opposed to being a datagram-based network. Nodes in the overlay network have switching state for each client session. This has the important implication that we can do service-level path failure recovery without waiting for the failure information to propagate and stabilize across the network. This allows us to scale in the dimension of the expanse of the system.

We use the emulation platform for an evaluation of our architecture. We perform a controlled design study to identify scaling bottlenecks. We find that the scaling bottleneck is the per-session switching state at the overlay nodes. Although the switching state allows us to restore service-level paths quickly, it also means that a large number of client sessions may have to be restored simultaneously on detecting a failure. However, we find that the provisioning require to handle this per-session switching state, to overcome this scaling bottleneck, is minimal. An off-the-shelf Pentium-III 500 MHz machine can easily handle the per-session switching state of at least 250 simultaneous client sessions. The additional provisioning is small in comparison to the provisioning required for the actual service components.

1.5.3 Load-balancing issues

The *performance* of a service-level path depends on both the load-level of the chosen service instances, and the performance of the underlying network path. We consider in depth the issue of load-balancing across the service clusters of our architecture. We introduce a metric for choosing a set of service instances for a composed client session: the *least-inverse-available-capacity (LIAC)* metric. This is used to assign costs to edges in a graph with service replicas at different nodes; this graph is a transformation of the overlay network graph. The least cost path (based on the LIAC metric) in this graph is chosen as the service-level path for the client.

An important aspect we study is the interaction between the metric for load-balancing and the mechanism for load information dissemination. This is the essence of the feedback loop for load balancing. This design study too is based on the implementation of our architectural components and the emulation platform. We first try a mechanism for load information dissemination based on periodic updates from the service replicas. Though this does well, we find that it causes load oscillations. We then introduce a *piggybacking* mechanism to update load information via the service-level path setup messages. This does not update load globally, but only along the service-level path, and has little additional overhead. Despite the fact that piggybacking updates load only along the service-level path, we find that it can achieve very good load balancing and can effectively reduce oscillations.

Piggybacking achieves good load balancing across replicas, but the LIAC metric often chooses far away service instances. This results in longer service-level paths, and hence in larger end-to-end latency for the client session. We introduce an additional factor in our LIAC load balancing metric – this achieves a good trade-off between length of service-level path and load balancing between service replicas. We find that this load balancing metric performs well under a variety of scenarios, including failure recovery of service-level paths *during* a client session.

1.5.4 Availability study using a practical application

The motivation for the operational model of service composition as we have considered it comes from personal mobility and service mobility scenarios in next generation heterogeneous networks. The *Universal Inbox* is an architecture for *extensible* any-to-any communication in a heterogeneous network. It provides the control mechanism for enabling personal mobility and service mobility in a wide variety of scenarios. As part of this thesis, we present how service composition forms a crucial part of the Universal Inbox architecture, and provides data-type independence in any-to-any communication integration. Our mechanisms for availability and performance in wide-area service composition fit in with the redirection and personalization components of the Universal Inbox.

From our implementation of the Universal Inbox, we choose a specific scenario involving a composed text-to-speech application. We use this to evaluate our architecture from the point of view of an implementation of the composed application. We first use the text-to-speech application in the emulation platform to study the effect of spurious failure detection timeouts on the application. We find that the overhead in terms of restoring application level soft-state during service-level path recovery is minimal.

We use a wide-area testbed to study the usefulness of our architecture in terms of its goals of availability improvement. We use the composed text-to-speech application for this study, and measure the improvement in availability due to our recovery algorithms. Without the use of the recovery algorithms, system availability is in the last percentage within 100% or even closer (over 99%

or even over 99.5% in many cases). But this is nowhere close to the five 9's availability standards set by telecommunication systems [52]. We find that the recovery algorithms can improve the system by pushing availability closer to 100%. We observed dramatic improvements in availability. In quantitative terms, we were able to reduce the system downtime by factors of up to 2-10 in many cases. This shows the usefulness of our architecture from the point of view of the end-user.

1.6 Thesis Outline

The rest of the thesis is organized as follows. The next chapter (Chapter 2) discusses related work. We present an outline of past research in several related areas in service composition, availability issues, and load-balancing in other contexts. Chapter 3 presents a feasibility analysis of failure detection on the wide-area Internet. This forms the basis of our architecture. We present our service-overlay-network based architecture for availability constrained and performance sensitive service composition in Chapter 4. We also present an emulation-based evaluation of the architecture to identify sources of overhead and scaling bottlenecks.

Chapter 5 focuses on the issue of load-balancing across the different service execution platforms of our architecture. We study the interaction between the load information propagation mechanism and the metric used for the choice of service instances for composition of client sessions.

In Chapter 6, we look at an application-centric view of service composition. We look at a class of application scenarios that involve service composition, and enable personal mobility and service mobility in a heterogeneous network scenario. We use a specific composed text-to-speech application to study the usefulness of our architecture in terms of improving the availability of the system.

Finally, we present concluding discussions and avenues for further interesting exploration in Chapter 7.

Chapter 2

Related Work

Research related to our work falls into two broad categories:

- Wide-area composition in contexts other than ours, and
- Availability, and load-balancing solutions for various kinds of services and networks.

Section 2.1 presents related work in the first category. Wide-area service composition so far has not considered performance and availability issues in data-interface based composition. We divide related work in the second category into three subsections. Section 2.2 presents various solutions to availability improvement of services, based on clusters of replicated servers. While these solutions improve availability by handling process- and machine-level failures within a cluster, they do not consider wide-area network path failures. We build on top of the *intra*-cluster failure handling mechanisms and improve availability further by specifically considering quick detection of and recovery from wide-area network path failures in the middle of client sessions. Section 2.3 considers solutions to load-balancing for servers replicated across the wide-area Internet. The work in this domain is focused on the choice of web-mirror replicas, and does not consider service composition. With composition, there are two novel aspects: (a) we have to choose a *set* of service instances and a service-level path, and not just a single web-mirror, and (b) we consider long-lasting client sessions and failure recovery in the middle of sessions. Finally, Section 2.4 presents approaches to improving network availability through the use of overlay networks. These overlay network scenarios do not involve service composition. With composition, we have to choose not just a simple overlay path, but a service-level path. A service-level path has constraints in that it has to contain a specific set of services in a specific order. This chapter concludes with Section 2.5, where we summarize the unique aspects of the problem domain and solution space explored in this thesis.

2.1 Wide-area composition

Object-oriented programming and the use of class hierarchies is a well known way to reuse software components and compose them for novel functionality. A merger of object-based systems with traditional Remote Procedure Call (RPC) [62] based client-server communication gave rise to distributed object based systems. Here, objects, which are programming language abstractions, are invoked, transported, and managed across the network.

The Object Management Group (OMG) [10], is a non-profit consortium created in 1989 with the purpose of promoting theory and practice of object technology in distributed computing systems. Originally formed by 13 companies, OMG membership grew to over 500 software vendors, developers and users. Common Object Request Broker Architecture (CORBA) [61] is OMG's standard specification for distributed object-based composition. The central architectural component of CORBA is the Object Request Broker (ORB). It consists of mechanisms to name, identify, and locate objects. It passes requests from clients to the object implementations on which the requested method exists. Language, hardware, and operating-system independent interface stubs are used for communication between the client and the server. The communication between the object implementation and the ORB core is effected by the Object Adapter (OA). It handles functionalities such as generation and interpretation of object references, method invocation, security of interactions, object activation and deactivation, mapping references corresponding to object implementations and registration of implementations.

The Distributed Component Object Model (DCOM) [4] is Microsoft's implementation of an object based system like CORBA, and is CORBA's main competitor. Java-RMI (Remote Method

Invocation) is a Java-centered mechanism for distributed object-based composition. Each of these have a different interface definition language for object interfaces, and wire-protocol for transferring objects.

Such composition falls into the category of program-interface based composition. CORBA, DCOM, and Java-RMI are focused on definition of object or service interfaces, wire-format for transporting objects from one machine to another, and programmatic handling of various erroneous conditions that might occur in a distributed system. In contrast, we are considering data-interface based composition where there is a data flow across the network. Composed client sessions could last for long durations of time. Our focus is on issues of availability and load balancing in the presence of scale.

Another important difference between traditional object-based distributed systems and composition of services in our case is borne out by the following perspective. In distributed systems, the CAP principle applies: "Strong **C**onsistency, High **A**vailability, **P**artition-resilience: pick at most two" [41]. That is, all three properties of consistency, availability, and partition-resilience are impossible to attain simultaneously. In systems like the Internet, network partition is inevitable, and hence one has to choose between consistency and availability. Traditional object-based systems for distributed computing choose consistency of object transactions, and hence have to sacrifice availability. In sharp contrast, we are dealing with service components that have only soft-state, and no persistent state. Application-state consistency is hence not a concern. We are thus able to improve on availability by building resilience to network failures in the middle of client sessions.

The web-services initiatives [80] extend object-based distributed composition to generic service components on the Internet. The operational model here is similar to ours and data-interface based composition is also considered. However, the focus is on the definition of service descriptions and interfaces. The Web Service Description Language (WSDL) [37] and Web Services Flow Language (WSFL) define a web service and a web-service composition respectively. They are XML based. WSDL defines how to access a service, while WSFL describes a composition by defining the "flow" of control and data in the composition.

The Open Grid Services Architecture (OGSA) [11] adapts web-services for scientific grid computing and addresses issues of service discovery, dynamic service creation, lifetime management of service instances, and manageability. The Meta Directory Service (MDS) is a directory of services, managed through soft-state registry. The public-key based Grid Security Infrastructure (GSI) provides single sign-on authentication, and communication protection. It can also be used for authentication with intermediate proxies in the data flow. GSI uses X.509 certificates.

The service description and interface language of the web-services domain, as well as the service management and security/authentication functionalities provided by OGSA are complementary to our architecture, since we focus on availability and performance issues in wide-area service composition.

The IETF Open Pluggable Edge Services (OPES) group [12] proposes a framework for "pluggable", or composable services [25]. The application space considered is similar to ours in that various content adaptation and transformation agents are considered in a data-flow based composition [24, 27]. However, they do not describe a wide-area architecture to address issues of availability or load balancing. Failure detection and recovery in the middle of a client session are not considered in this framework. The OPES framework includes the Intermediary Rule Markup Language (IRML) [26], which is an XML-based language for specifying rules for modification of content by intermediary services. For instance, a content source like CNN can specify the parts of its web-page content that can be adapted for a thin client. This rule definition language is orthogonal to the issues of availability and performance, and is complementary to our work.

Another research effort that considers composition across the network is Application Layer Active Network (ALAN) [42, 43]. Application layer routing of data is done through *proxylets*. Proxylets are the service components that are composed with content sources, in a data interface based composition. They are proxies that are dynamically deployed from proxylet servers, as needed by client sessions. Dynamic Proxy Servers (DPS) are the places where the proxylets are run.

The operational model in ALAN is different from ours in that the proxylets can be dynamically created and hence fall under the category of mobile code. In our operational model, the services are deployed and managed by different service providers at specific locations, and the code is not mobile. The services could be heavy-weight in nature, and service providers maintain control over their code. This reflects the way in which services are deployed on the Internet today. ALAN focuses on the functionality of dynamically creating and executing the proxylets. Whereas, we focus on the availability and performance aspects. Quick recovery from failures is one of our goals, and we specifically evaluate this aspect of our system.

2.2 Cluster-based availability

Availability of web-enabled services is an important concern when the service is being accessed by clients across the Internet. Scaling to a large number of clients is a crucial consideration as well. Cluster-based platforms have emerged as an effective method to address the issues of availability and scale. A cluster consists of a number of, often homogeneous, off-the-shelf machines that are interconnected via a high-speed local area network (LAN) [9]. This is depicted in Figure 2.1.

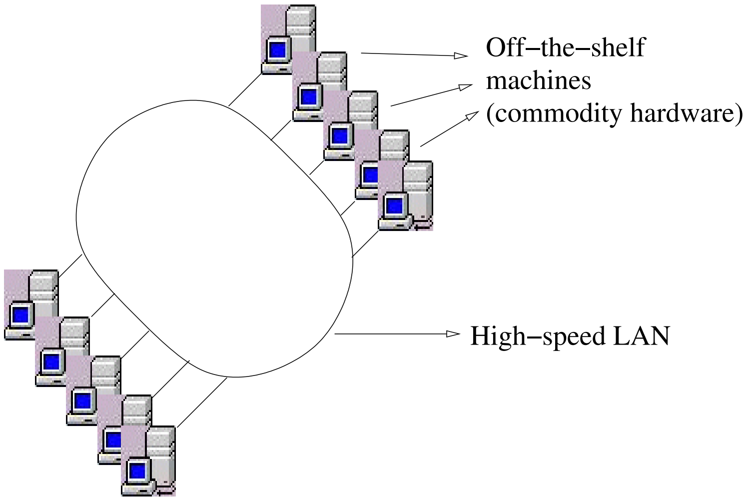

Figure 2.1: Cluster of workstations

The use of a cluster of workstations as opposed to using larger and more powerful single machines to scale a system has several advantages. The cluster can be incrementally provisioned to accommodate an increasing client population. The use of commodity hardware reduces overall cost.

Software upgrades are made easy, by bringing down parts of the cluster at a time and installing changes incrementally. Importantly, if a service is replicated across the machines in a cluster, the overall availability can be improved by handling process- and machine-level failures of the server.

Mechanisms for handling such failures have been developed for a variety of services. For web-servers (HTTP servers), a level-4 switch placed in front of the cluster can redirect incoming client traffic to any server replica, based on the liveness and load-level of the server. Cisco's LocalDirector [3] is a commercially available product that does such redirection. Mechanisms for intelligent redirection of client requests have also been researched. The LARD project (Locality Aware Request Distribution) [63] explored mechanisms for client redirection based on the locality of the requested web-page.

With the growth of the number of Internet clients, there was also growing heterogeneity in the capabilities and connectivities of clients. Proxies evolved as a mechanism for adapting content across this heterogeneity. The use of proxies is a form of composition at a coarse granularity, since the proxies are rather independent of the servers or the clients. This is a data-flow based composition as there is data flow from the server to the proxy to the client.

The TACC project (Transformation, Aggregation, Caching, and Customization) [40] developed cluster-based mechanisms to achieve availability and scalability of proxy components. Availability is improved through handling process/machine-level failures. A cluster-manager monitors the liveness of the services and a front-end dispatches incoming client requests to the least loaded service instance.

TACC also considered the composition of proxy components. However, the availability mechanisms developed as well as the composition considered was within the single cluster. Monitoring and load balancing within the cluster is relatively simple due to the high-speed, low-latency network. Although the TACC project considered network partitions within the cluster, such partitions can be minimized to a great extent by means of redundancy and careful engineering of the local network. However, this is not possible in the wide-area network where the path could span multiple administrative domains. TACC does not consider failure of connectivity between the server and the cluster, or between the cluster and the client. In other words, wide-area network path failures are not handled. Cluster-based web-server solutions also do not handle this issue. We leverage the *intra*-cluster availability improvements achieved by TACC and complement it with *inter*-cluster mechanisms for availability improvements through the handling of wide-area network path failures.

The Active Services (AS1) model [19] provides a cluster-based mechanism for handling failures of client-side proxies. The solution is based on a multicast soft-state heart-beat from the client to the service cluster. This heart-beat instantiates *servents*, or service agents, as necessary. AS1 does not consider composition of services across the wide-area Internet. Like in the case of cluster-based availability solutions, AS1 does not consider issues of wide-area network path failure detection or recovery.

2.3 Load-balancing across the wide-area

In the context of web-servers, there has also been a plethora of research on load balancing and improvement of availability by using replicas distributed across the wide-area Internet. A variety of mechanisms have been studied for load balancing of distributed web-server systems [33]. These include:

- *Client-side approaches:* Here, the redirection to an appropriate server is done at the client-side, either by the browser itself, or by a client-side proxy. In Smart Clients [83], an applet is first transferred from a server replica to the client. The applet then sends messages to the server replicas to learn the appropriate server instance for subsequent requests. In [23], and in SPAND [76], a client-side proxy maintains information about the load on the server replicas, and redirection is done using this information.

- *DNS-based approaches:* The client request is redirected to a particular server replica by returning an appropriate IP-address mapping during the DNS lookup by the client. A simple round-robin approach was implemented by National Center for Supercomputing Applications (NCSA) [53]. The SunSCALR framework enhances this with a mechanism to exclude unreachable or highly-loaded servers [77].

- *Dispatcher-based approaches:* This extends the address virtualization done by DNS-based approaches at the URL-level, to the IP-level. A single virtual IP-address is given to the distributed web server system, and a dispatcher rewrites packets at the IP level while redirecting to an appropriate server. An example of such a system is the IBM Network Dispatcher [45].

- *Server-based approaches:* Here the redirection is done by the originally chosen server. For example, *Scalable server World Wide Web* (SWEB) [21] includes a HTTP redirection mechanism. In *Dynamic Packet Rewriting* (DPR) [29], the redirection is done by packet rewriting at the originally chosen server.

Choice of service instances for service composition involves at least two novel aspects over traditional web-server selection:

- First, unlike web-mirror selection, we have to choose a *set* of service instances for each client. We have to choose a service-level path, and not just a single web-server instance. We need to ensure connectivity between the service components in the composition.

- Second, we consider failure detection and recovery of composed service in the middle of a long-lived session. Web downloads usually last for a short duration, and hence recovery during a client session has not been a consideration in web-server selection mechanisms.

These lead to a consideration of a fundamentally different architecture that consists of an overlay network of service clusters over which services are composed.

2.4 Improving network availability through overlay networks

Research has shown that the default IP path is often times not optimal [74]. In [74], it was found that in 30-80% of the cases, there is an alternate path with significantly superior quality in terms of reachability as well as end-to-end latency. With this idea in mind, overlay networks have emerged as a possible mechanism to get around deficiencies in the underlying Internet path. An overlay network is built on top of the IP layer.

The RON (Resilient Overlay Networks) project [20] proposes such an overlay network built on top of the IP layer. A network of RON nodes is built as a complete graph and these nodes exchange routing and performance information. RON proposes overlay networks of a small number of nodes (20-50), built for application specific purposes. The overlay network provides overall better performance than the underlying Internet by using an overlay path when the default Internet path performs poorly or fails.

In [22], a mechanism for video delivery using a pair of Internet paths simultaneously is proposed. This is achieved by an application level overlay path in addition to the default Internet path. This is shown to achieve significantly better video performance.

Content-addressable networks build a network-wide distributed hashtable using an overlay network of nodes. The primary purpose of the hashtable is to locate named objects in the network. CAN [69], Tapestry [86], and Chord [79] are three such systems – they differ in the way the distributed hashtable is built and maintained. The overlay network in these systems provides opportunities for using alternate paths to locate objects if the original path experiences a failure. However, since routing is based on the named object, and not on the shortest possible path, these networks can have high routing stretches. This can be up to factors of 2-4 even in the overlay network [86].

In all these systems, service composition is not considered. In service composition, we require not just a simple network path, but a service-level path. A service-level path is a network path with "constraints" on it – constraints in terms of a particular set of services on it in a particular order. This requirement leads to an altogether different architecture. We also consider *quick* recovery for real-time applications as one of our goals. This has not been considered in the design of other overlay networks. Our architecture is different from other overlay networks in that ours is a service-level overlay, and routing on it is *virtual-circuit* based, as opposed to being datagram-based. This allows us to effect recovery without having to wait for the failure information to propagate and stabilize across the network.

2.5 Summary

We summarize our comparisons with related work in Table 2.1: past work that has considered composition has not addressed the wide-area availability and performance issues, and work that has considered availability and performance issues of servers or network paths has not considered service composition. The issues of availability, performance, and scalability have not been addressed in the context of composition of services across the wide-area Internet.

The rest of this thesis presents our solution for availability and performance in wide-area service composition. In the next chapter, we present our analysis of Internet path behavior to answer the questions with respect to failure detection. This forms the basis of our architecture presented subsequently.

	Service composition	**Availability considerations**	**Performance, load-balancing**	**Other remarks**
Other wide-area composition:				
CORBA, DCOM, JavaRMI	Program-interface-based composition	Availability sacrificed in return for consistency of distributed objects	Not addressed	
Web-services (WSDL, WSFL) OGSA (MDS, GSI)	Yes	Not addressed	Not addressed	Service-interface description, service mgt., security fnlty. are complementary
IETF OPES	Yes	Not addressed	Not addressed	
ALAN	Yes, using proxylets (mobile code)	Not addressed	Not addressed	Different operational model (mobile code)
Cluster-based availability				
Web-server solutions Cisco's LocalDirector, LARD	No	Addresses only intra-cluster failures	Within cluster	Intra-cluster failure handling is complementary
TACC	Only within cluster	– same –	– same –	– same –
AS1	Restricted: only one client-side proxy	– same –	– same –	– same –
Load-balancing across the wide-area				
Web-server load balancing,	No	Quick failure recovery not considered	Yes, across wide-area replicas	For composition, a *set* of servers have to be chosen
Improving network availability using overlay networks				
RON	No	Yes, through use of alternate paths	Issue does not arise, since services are not considered	Overlay network: datagram-based, a full-graph, of limited size
CAN Tapestry, Chord	No	Yes, but quick recovery not a consideration	– same –	Routing on these can have a large routing stretch
Our architecture for wide-area service composition	Yes	Yes, through quick failure detection and recovery in the wide-area	Yes, using graph metric based algorithm on the overlay network	Service-level overlay network; virtual-circuit based

Table 2.1: A comparison with related work

Chapter 3

Internet Failure Behavior and Failure Detection

One of the important goals in our work on wide-area service composition is high availability from the point of view of the clients of composed sessions. Recall that a service-level path consists of a set of service instances and the network path in-between. Hence the availability of the system could be affected in two ways: by failure of a service instance in the service-level path, or by failure of the network path that constitutes a leg of the service-level path. This is shown in Figure 3.1.

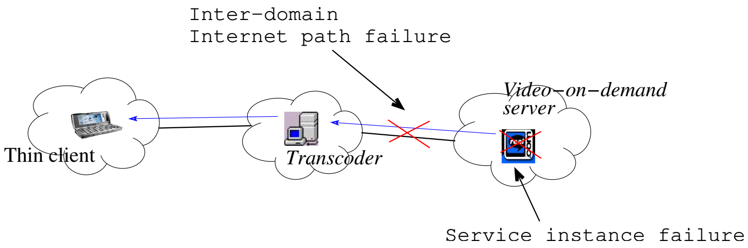

Figure 3.1: Network path failures, and service instance failures

Past research has considered availability improvements through the use of cluster platforms for handling server failures (e.g. TACC [40]). Although quick recovery has not been a consideration in TACC, it is conceivable that "hot" backups of service instances can be used within the cluster to achieve this. Cluster platforms can be tightly controlled centrally and due to the presence of a reliable, high-speed, low-latency LAN, process- or machine-level failures can be detected and recovered from within a few hundred milliseconds.

In our work, we consider the issue of detecting and recovering from the second kind of failures in the service-level path. We wish to detect network failures in the service-level path and recover from them as quickly as possible so that the application sees minimal or no interruption. In particular, we are concerned about keeping track of the liveness of the wide-area Internet path between successive components in the service-level path.

There are several crucial questions to answer in this respect:

- First, is there a clearly defined notion of failure, or long lasting outage? The network between successive components in a composition could be an inter-domain path. There is inherent variability in delay, loss-rates, and outage durations on an inter-domain Internet path. We have used the term "failure" loosely so far. It is an open question as to whether there is a clearly defined notion of failure. Intermittent congestion could last for varying periods of time. There could also be intermittent short outages due to a variety of factors (such as intra-domain route change, router reboot, etc.). In such a case, a clear definition of failure becomes difficult.

  For real-time streaming applications such as those in the examples in Section 1.1, we consider Internet path outages such as those that happen when there is a BGP-level failure [54]. Such path outages could last for several tens of seconds to several minutes. We wish to detect such long outages. In the rest of the discussion, we use the terms "failure" and "long-outage" interchangeably, and both refer to instances when no packet gets through from one end of the Internet path to the other for a long duration such as several tens of seconds.

- Second, if there is a notion of failure, how quickly can this be detected? It is important to ensure that failure detection and recovery is *quick* for real-time applications. Ideally, recovery within a few hundred milliseconds is suited for interactive applications. Recovery even within the limits of a few small number of seconds (3-4 seconds) would be very useful for applications. This is especially so for non-interactive applications such as streaming of stored video, which usually have buffers of 5-10 seconds. Detection and recovery within these bounds can potentially hide long-lasting inter-domain outages from the end user, thus improving the overall availability.

- The third question, given a failure detection timeout, how often is intermittent congestion or short outage confused for a failure?

In short, our objective is to determine *how quickly (within a few small number of seconds) failure detection can be done with minimal overhead in terms of spurious failure detection timeouts.*

The above issues are relatively straightforward with respect to the first kind of failures – service instance failures within a cluster. Tight control on the machines is possible, and latencies are small. Quick failure detection is especially difficult to achieve given that there is no support for such detection from the underlying Internet. Since we are considering inter-domain paths that span multiple ASes (Autonomous Systems), such support is especially difficult to achieve. Commercial ASes do not like to share or reveal information about their internal network structure, protocols used, outage patterns, etc.

The second and third questions posed above are related, and involve a trade-off. Intuitively, an aggressive failure detection mechanism with a small timeout may trigger *spurious* path restorations, where we confuse intermittent congestion or short outages in the Internet path with a failure. On the other hand, a conservative failure detection mechanism with a large timeout could mean longer detection times in general.

3.1 Heart-beat based monitoring

Given that we do not have support for failure detection from the underlying Internet, the straightforward way to monitor for liveness of the network path between two Internet hosts is to use a keep-alive heart-beat, and a *timeout* at the receiving end of the heart-beat to conclude failure. This is shown in Fig. 3.2. There is a notion of a *false-positive* when the receiver concludes failure too soon, when the outage is actually not long-lasting (Fig. 3.2). False-positives occur due to intermittent congestion/loss or other short outages. We term a path restoration triggered by such a false-positive to be a *spurious* path restoration.

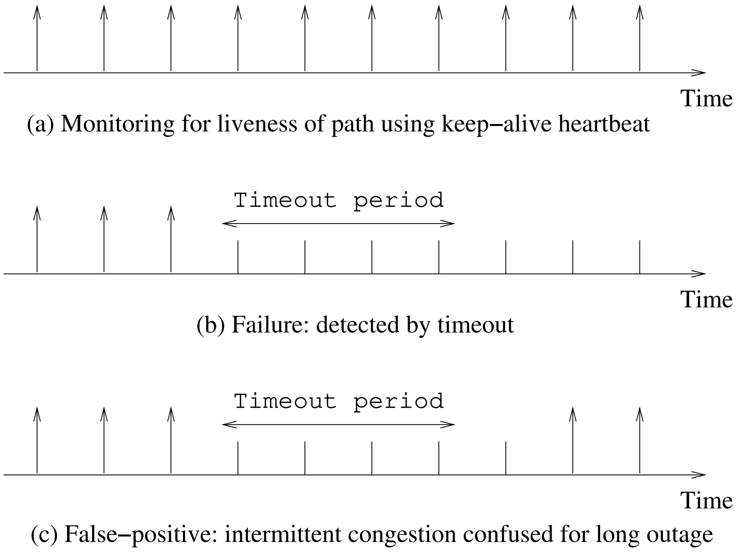

Figure 3.2: Failure detection using heart-beats

Note that the failure detection is done by the downstream node, and that we only rely upon a one-way heart-beat stream for failure detection. This in turn implies that we do not need to assume symmetry in the underlying one-way Internet paths between the source and destination. (Since failure detection is done downstream, this suggests that the failure recovery process be initiated downstream – we return to this in Chapter 4).

There are three questions to answer in this context.

- **Q1:** What should be the heart-beat period?
- **Q2:** Given a heart-beat period, what should the timeout be to conclude long outages?
- **Q3:** Given a timeout period, how often do false-positives occur, when we confuse intermittent congestion or short outages for a long outage?

These questions are not independent of one another, but are closely related. They are also related to the original three questions we posed earlier, and help us answer them.

To decide the period of the heart-beats, we make the observation that we are interested in detecting long-lasting outages *quickly*, within a few small number of seconds. Hence it does not make sense to choose a heart-beat period greater than a second or more. For instance, a heart-beat period of 1 second with a timeout of 2 seconds does not make sense. We choose a heart-beat period significantly smaller than a second. We empirically choose 300 ms to be this value. (The reason why a more frequent heart-beat will not be significantly more useful will be apparent after we present the trace data below).

3.1.1 Trace data

To answer the other two questions posed above, we need a frequency/probability distribution of the incidence and duration of outages. There have been studies of outages or packet loss patterns at small time scales (less than 1 sec) [82, 30]. These have shown that there is correlation of packet loss behavior within one second, but little correlation over a second. (We have used this above, in our determination of the heart-beat period). Further studies have estimated failures that last for over 30 sec [85, 34]. To the best of our knowledge, there does not exist publicly available data, or a study, that gives a probability distribution of these failure gap periods on a wide-area Internet path.

We have collected data to arrive at such a probability distribution. We run a simple UDP-based periodic heartbeat between pairs of geographically distributed hosts. We choose a heart-beat period of 300 ms, as explained above. The set of hosts from which we collected data are: Berkeley, Stanford, CMU, UIUC, UNSW (Australia), and TU-Berlin (Germany). This represents some trans-oceanic links, as well as Internet paths within the continental US (including Internet2 links). We have data for nine pairs of hosts among these, a total of 18 Internet paths. Six of the nine pairs of data were collected in Nov 2000, and three in Oct 2001. One pair of hosts was a repeat between these two runs. Across these eighteen paths, the RTT varied from about 3 ms to 335 ms. The number of AS domains on each Internet path varied between 3 and 7 for these eighteen paths. The heart-beat exchange was done for an extended period of time – for 3-7 days for the 9 pairs of hosts. The Internet paths considered in this study, and their various properties are given in Table 3.1.

HB destn	HB src	Total time	Base RTT (approx.)	# ASes in path
Berkeley	UNSW	130:48:45	220 ms	6
UNSW	Berkeley	130:51:45	220 ms	6
Berkeley	TU-Berlin	130:49:46	170 ms	6
TU-Berlin	Berkeley	130:50:11	170 ms	6
TU-Berlin	UNSW	130:48:11	335 ms	7
UNSW	TU-Berlin	130:46:38	335 ms	7
Berkeley	Stanford	124:21:55	3 ms	3
Stanford	Berkeley	124:21:19	3 ms	3
Stanford	UIUC	89:53:17	50 ms	5
UIUC	Stanford	76:39:10	50 ms	5
Berkeley	UIUC	89:54:11	52 ms	5
UIUC	Berkeley	76:39:40	52 ms	5
Berkeley	CMU	168:19:42	56 ms	5
CMU	Berkeley	168:19:32	56 ms	5
CMU	Stanford	168:14:19	55 ms	5
Stanford	CMU	168:14:52	55 ms	5
Stanford(2)	Berkeley(2)	168:12:17	3 ms	3
Berkeley(2)	Stanford(2)	168:12:08	3 ms	3

Table 3.1: Internet paths used in heart-beat experiments

3.1.2 Analyzing the trace data

To understand the nature of Internet path outages, we compute the gaps between successive heart-beats at the receiving end. This allows us to answer Q2 and Q3, as we explain below. Looking across all gap-lengths in an experiment, we get a distribution. We plot this as a Cumulative Distribution Function (CDF). Fig. 3.3(a) shows such a CDF for 3 pairs of hosts (six Internet paths). The plots for other host-pairs are similar and we do not show them here.

Note that the y-axis in Fig. 3.3 starts from 99.9%. This is because a large number of gaps in reception that are between 300 ms and 600 ms. This is merely inter-arrival jitter since the heart-beat period itself is 300 ms. In the graph, we first draw attention to the last plot which is marked as the ideal case. This is with fictional data, has no connection with our trace data, and is for purposes of illustration. We term this plot as ideal since there is a long flat region in the CDF. This flat region starts from 1,800 ms and continues up to 30,000 ms (30 sec), before the CDF begins to increase again (this increasing part beyond 30 sec cannot be seen on the graph). This

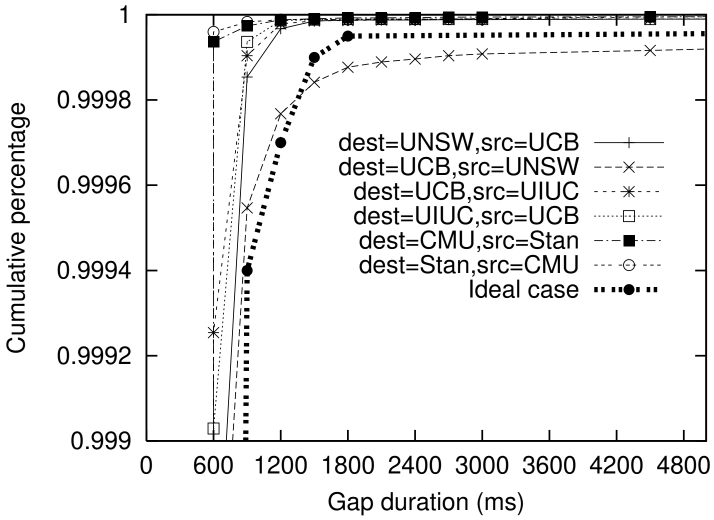

Figure 3.3: Gap distribution (CDF)

long flat region means that, if we choose a timeout of 1,800 ms for detecting long outages, we would never confuse an intermittent congestion or short outage for a failure. That is, all intermittent/short outages last for less than 1,800 ms. An outage lasting 1,800 ms implies an outage lasting for longer than 30,000 ms.

This effectively answers Q2 and Q3 for ideal case. Answer for Q2: a failure detection timeout of 1.8 seconds. Answer for Q3: intermittent congestion or short outages are never confused with a long outage.

We observe that the plots with the real data are very close to the ideal case. There is a sharp knee in the plot, and the CDF has a region that is almost flat beyond this knee. This suggests a value for the timeout (for failure detection) that is just beyond the knee in the plot. For the different plots, this value varies between 1,200 ms and 1,800 ms. This tentatively answers Q2 (tentatively, since Q2 and Q3 are inter-dependent).

For the real data, the region beyond the knee is “almost” flat, but it definitely has a small slope in all the plots with the real data. This slope means that there is a non-zero probability that we confuse intermittent congestion or short outages with long outages or failures. That is, there is intermittent congestion or short outages that last for periods of time ranging beyond the timeout

HB destn	HB src	Total time	Num. False Positives	Num. Failures	False-pos. Perc.	Failure Pred. Prob.
Berkeley	UNSW	130:48:45	135	55	71%	29%
UNSW	Berkeley	130:51:45	9	8	53%	47%
Berkeley	TU-Berlin	130:49:46	27	8	77%	23%
TU-Berlin	Berkeley	130:50:11	174	8	96%	4%
TU-Berlin	UNSW	130:48:11	218	7	97%	3%
UNSW	TU-Berlin	130:46:38	24	5	83%	17%
Berkeley	Stanford	124:21:55	258	7	97%	3%
Stanford	Berkeley	124:21:19	2	6	25%	75%
Stanford	UIUC	89:53:17	4	1	80%	20%
UIUC	Stanford	76:39:10	74	1	99%	1%
Berkeley	UIUC	89:54:11	6	5	55%	45%
UIUC	Berkeley	76:39:40	3	5	38%	62%
Berkeley	CMU	168:19:42	108	4	96%	1%
CMU	Berkeley	168:19:32	46	4	92%	8%
CMU	Stanford	168:14:19	12	3	80%	20%
Stanford	CMU	168:14:52	17	4	81%	19%
Stanford(2)	Berkeley(2)	168:12:17	50	1	98%	2%
Berkeley(2)	Stanford(2)	168:12:08	101	0	100%	0%

Table 3.2: Occurrence of false-positives

value. To give an quantitative idea of this observation, suppose that we want to detect outages lasting for 30,000 ms or more, and have a timeout of 1,800 ms. For four of the eighteen Internet paths, the timeout would be able to predict a long outage with probability 40% or more. For six other cases, this probability would be between 15% and 40%, and for the rest of the eight cases, the prediction probability would be less than 15%. These percentages are summarized for the 18 Internet paths in Table 3.2. (These percentages cannot be read off the graph; we computed them directly from the data used to plot the graph).

This partially answers Q3. The cases where the timeout wrongly predicts a long-lasting outage are exactly the cases where intermittent congestion or short outages occurs. However, the relative rate of occurrence of false positives is not as important as the absolute rate of occurrence. For this, we plot a second set of graphs in Fig. 3.4. This shows the rate of occurrence of outages of various durations on a log scale. If we consider outages of 1,800 ms or above, these occur about once an hour or less frequently. (The long outages happen even less frequently, but contribute significantly to loss of availability). Intuitively, this is a very small absolute rate of occurrence of timeouts, and hence a small rate of occurrence of false positives.

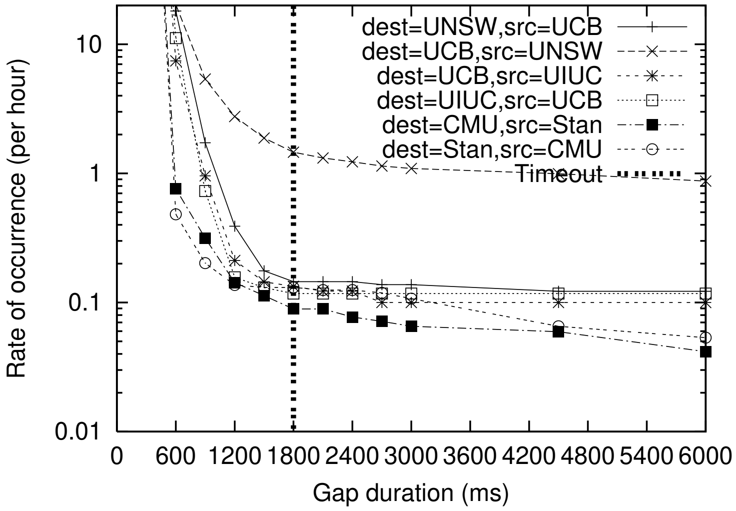

Figure 3.4: Outage occurrence rate

This then effectively answers Q2 and Q3: for a timeout value just beyond the knee in Figure 3.3 (1,200-1,800 ms), the rate of occurrence of false positives is as given in Figure 3.4.

The plots in Fig. 3.4 also have knees at around the same points as those in Fig. 3.3. This also explains why a timeout just beyond the knee (in either of these plots) is appropriate. A value before the knee, say 1,000 ms, for the timeout would mean that timeouts occur much more frequently – 1-2 orders of magnitude more frequently. This in turn implies a correspondingly large absolute rate of occurrence of false positives. On the other hand, a timeout value much beyond the knee, say 3,000 ms does not bring much reduction in terms of rate of occurrence of false positives, but only increases the failure detection time significantly. We also note that although the prediction probability of long outages with a small timeout (1,200-1,800 ms) may be poor (Table 3.2 shows values less than 15% in many cases), it really does not matter since the absolute rate of occurrence of false positives is small.

Finally, we also observe that since we are using a continuous stream of heart-beat losses, a more frequent heart-beat (than once in 300 ms) is unlikely to be more useful. For instance, a 50 ms heart-beat period is likely to be as good as a 300 ms heart-beat period. This is because, a stream of 300 ms heart-beat losses lasting for about 1.8 sec, or a stream of 50 ms heart-beat losses lasting for a similar period of time, could both have been caused most likely only by a long-lasting outage.

3.1.3 Statistical analysis of time-correlation

In the above discussion, we have not said anything about the time-correlation of the data. If the timeout events, for a particular value of the timeout can be shown to exhibit time correlations, then it is conceivable that an adaptive timeout mechanism can be designed, rather than choosing a fixed timeout. For instance, if timeouts are known to be correlated within, say 5-minute periods, then the timeout value can be slowly increased on detecting frequent timeouts within a 5-minute period. We consider this aspect of the analysis through statistical methods.

Our analysis proceeds as follows. We first plot the inter-arrival times of failure events in a time-series. This indicates that failure events may be correlated in time. We then plot the auto-correlation function at various lags, as well as the spectral density plots of the time-series of failure events. These plots further indicate possibilities of time-correlation in several of the traces. We finally perform the Box-Ljung Q-statistic test [57] on the time-series of timeout events. We find strong evidence against the hypothesis that the timeout events are IID. This suggests possibilities of designing an adaptive timeout mechanism. However, for the purposes of our architecture design, we consider only the simple, fixed timeout mechanism. The details of the statistical analysis summarized above can be found in Appendix A.

3.2 Summary

Our trace data has helped us answer the questions with respect to failure detection that we started out with. It is possible to define a notion of failure, or long-lasting outage on an inter-domain Internet path, without relying on the underlying Internet for any support. It is possible to detect these long-lasting outages (a) with a heart-beat mechanism with a period of about 300 ms, (b) with a failure detection timeout of about 1,200-1,800 ms, and (c) with a small absolute rate of occurrence of spurious timeouts, or false-positives (less than once an hour).

Although the ideal timeout is likely to be different for different Internet paths, from our data spanning a range of Internet paths, we can say that this value is likely to be in the 1-2 second range (1.2-1.8 sec in our data) in most cases. This value can be dynamically computed in a long running operational system.

These results with respect to failure detection encourage us to design an architecture around the failure detection mechanism for highly available wide-area service composition. If a quick recovery mechanism can be built on top of the quick failure detection mechanism, it would be really useful for composed real-time applications. By effecting failure detection and recovery within a few small number of seconds (3-4 sec), we would be able to avoid long lasting outages (several tens of seconds to several minutes) that happen during BGP failures. In the next chapter, we turn to the design of our architecture that builds around the failure detection mechanism.

Chapter 4

An Architecture for Wide-Area Service Composition

In this chapter, we present the design of our architecture for highly available wide-area service composition. We build on top of the failure detection mechanism presented in Chapter 3, and design mechanisms to perform recovery after failure detection. Our architecture achieves availability through quick recovery of service level paths, and performance through load balancing among the various service replicas. We consider scalability as an important goal in the design of our architecture.

This chapter is divided in three main parts. We first present the design of our architecture in Section 4.1. This motivates the development of a network emulation platform for an evaluation of our system. We describe our emulation testbed in Section 4.2. Subsequently, Section 4.3 presents our experiments to evaluate our architecture.

4.1 Design

4.1.1 Goals, requirements, and challenges

The primary goals of our architecture are:

- *Availability:* system downtime, or unavailability should be minimized – composed services should be highly available to users/clients. Real-time composed sessions should see minimal or no interruption due to failures. We wish to achieve availability through handling various kinds of failures, and recovery using alternate service-level paths and alternate service replicas.

- *Performance:* through appropriate choice of service instances and service-level paths, during path creation as well as recovery.

We wish to achieve these in the presence of scale:

- *Scalability,* when the system is deployed and in use. We wish to achieve scalability in several dimensions:
  - Number of clients
  - Expanse of the system: by this, we mean that the the system should operate on an Internet-wide distributed scale. The system should be potentially usable for clients on all parts of the Internet.
  - Number of service instances

With respect to our goal of availability, we consider the metric of *time-to-recovery* of client sessions on experiencing failure. This reflects in the overall reduction in system downtime, expressed as a percentage of the total time; this is the improvement in availability. With respect to our goal of performance, we focus specifically on load-balancing across server replicas as a metric and consider the load variation in time. We consider scaling in the various dimensions in the presence of the above metrics of availability and performance.

Given the above goals, we have the following requirements:

- *Liveness tracking and failure detection:* We need to track the reachability between the different service components, in order to know which instances can be chosen for composition.

- *Performance information collection:* We need to know about the performance and load-levels of the various service instances so that the least-loaded ones can be chosen.

In addition, we also have the following requirement for service composition:

- *Service location:* We need to know where the various service instances are located.

The choice of service instances for service-level path creation/recovery is somewhat like web-mirror selection, but is more challenging for at least two reasons:

- In general, we may need to select a *set* of instances for a client session, and not just one web-server.

- Unlike traditional web-server selection mechanisms, client sessions in our scenario could last for a long time, and it is desirable to provide mechanisms for path recovery using alternate service instances *during* a session.

These point to a further requirement. *Global information* about reachability and performance is needed. A hop-by-hop approach where each leg of the path is constructed independently could result in sub-optimal paths – a good choice of the first leg of the path could mean a poor choice for the second leg. Or, it may even happen that no instance of the required second service for composition is reachable from the first chosen service replica. An example of such a scenario is shown in Figure 4.1. Hence, a simple architecture that uses such a hop-by-hop approach is not appropriate for ensuring availability and performance in wide-area service composition. We reject this approach.

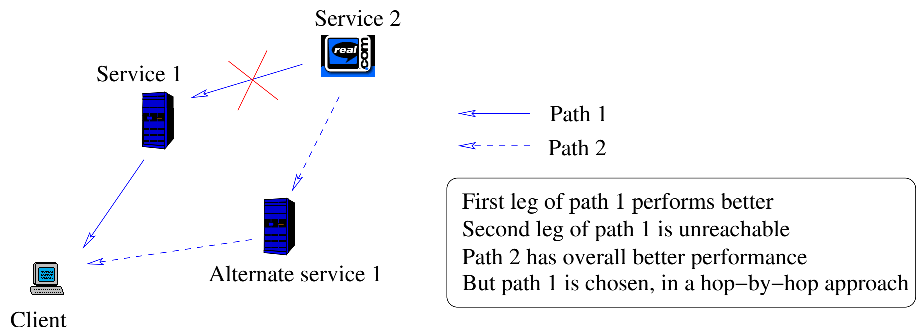

Figure 4.1: Hop-by-hop composition

Recovery is especially challenging when the system scales in the number of clients. Recall that sessions could be long-lasting, and hence the number of *simultaneous* client sessions grows with the number of clients. This means that a large number of client sessions may have to be restored simultaneously when a failure is detected.

Failure detection and recovery are challenging when the system scales to a large expanse and has a large number of service replicas. Collecting and maintaining current network reachability and performance information between service component instances becomes harder.

4.1.2 Architecture

Since service components are central to composition, we think in terms of service-execution platforms, and a service-level *overlay network*. Our architecture for composition is depicted in Fig. 4.2. We have three planes of operation: at the lowest layer is the hardware platform consisting of compute clusters deployed at different points on the Internet. This constitutes the middle-ware platform on which service providers deploy their services. Providers could deploy their own service cluster platforms, or could use third party providers' clusters. We define a logical overlay network on top of this. At the top-level, service-level paths are constructed as paths in the overlay network.

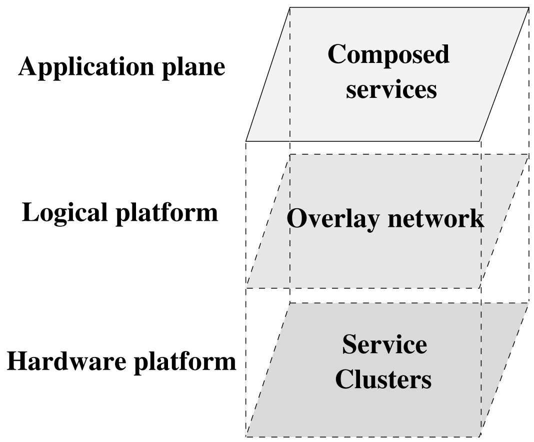

Figure 4.2: Architecture: Three layers

The use of cluster execution platforms as building blocks is an important design feature.

The rationale behind this is multi-fold. First, we wish to leverage known mechanisms to handle process- and machine-level service instance failures within each cluster execution platform [40]. This allows us to focus on mechanisms for handling wide-area network path failures. Second, clusters form an ideal execution platform for service components. Compute clusters can be provisioned and managed by the same service provider as the one responsible for the software service component. Alternatively, compute clusters can be deployed and managed by third party entities, for various service component providers to come and deploy their service instances. These scenarios fit in well with our operational model for service composition where there are multiple independent service providers.

The rationale behind the use of an overlay network is also multi-fold. First, the overlay network removes the reliance on Internet path recovery for availability of service-level paths. This is important since we do not depend on the underlying Internet for quick failure detection or recovery. Much as in [20, 22, 86, 69, 79], the overlay network allows us to define alternate service-level paths to achieve recovery on failure. The overlay network provides the context for keeping track of the reachability information between the various service components. There are however important differences in our overlay network compared to the ones in [20, 22, 86, 69, 79]. These differences stem from the fact that ours is a service-level overlay network. We point out these differences as we proceed with our discussion.

In addition to allowing us to keep track of reachability information, the overlay network also allows us to track the performance of the various service execution platforms. This is very important for balancing load across the various service instances.

Fig. 4.3 shows the architectural components as they would be deployed on the Internet. Each oval in the figure represents a service cluster execution platform. Each cluster has one or more independent service components. A service-level path is formed as a path in the overlay network. An example is shown in the figure, using an instance each of "Service 0" and "Service 1", in that order. Each cluster also implements a (trivial) "no-op" service that simply provides data connectivity, and does not perform any operation on the data. These no-op services allow composition of services that

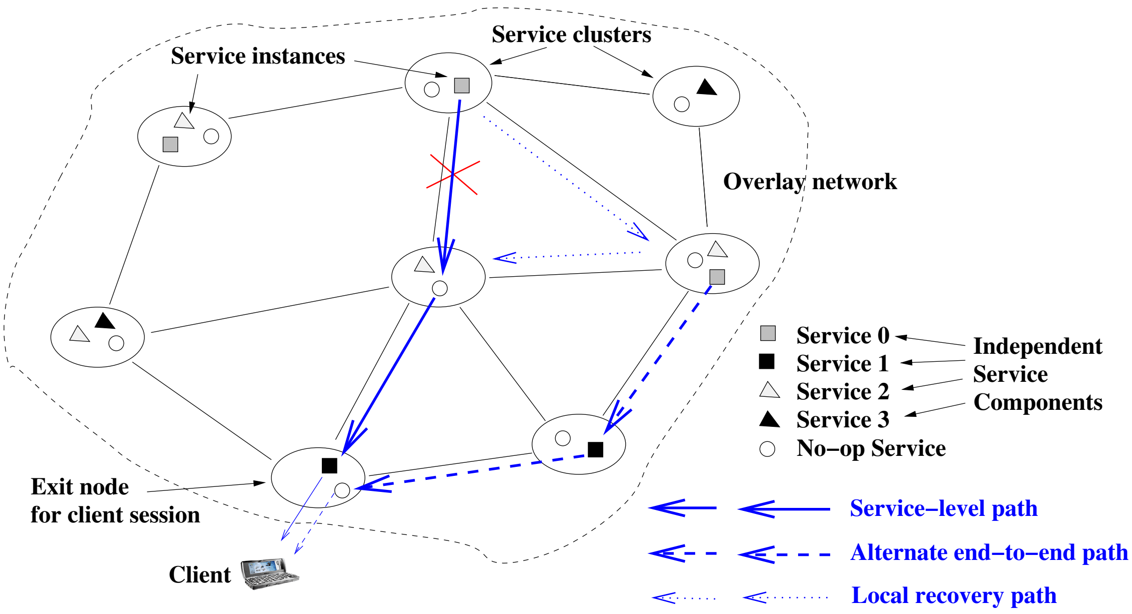

Figure 4.3: Architecture

are not necessarily adjacent in the overlay network. The dotted lines in the figure illustrate how the overlay network can be used to effect service-level path recovery on failure detection.

The use of cluster execution platforms has several advantages:

- As mentioned earlier, it allows us to separate process- or machine-level failures from wide-area network path failures. We term this as *hierarchical monitoring*.

- Another important advantage is that, with the use of clusters, the overhead of monitoring the liveness of an Internet path representing an overlay link, as well as the overhead of maintaining the distributed overlay graph state, are amortized across all client sessions and all service instances. That is, these overheads are neither dependent on the number of client sessions nor on the number of service instances in deployment.

- Third, the system now has two dimensions in which it can grow: the number of clusters, or the size of each cluster. This is an important feature that allows us to separate the issue of scaling in the dimension of the expanse of the system from the issue of scaling in the number of clients or number of service instances. As the number of clients or the number of service instances grows, the system can be provisioned additionally by expanding each cluster, rather than growing the number of overlay nodes.

We now divide the remainder of our discussion in this section into three parts. We first describe the various software functionalities and their interaction to enable composition, in Section 4.1.3. Then we discuss the important aspect of the scale and extent of the service overlay network, in Section 4.1.4. In the context of these discussions, Section 4.1.5 brings out the various aspects of the system that call for quantitative evaluation.

4.1.3 Software functionalities

For each composed client session, the data exits the overlay network, after passing through the required set of services. The overlay node at which the data exits is called the *exit-node* for that particular client session. The exit node is the one that interfaces with the client, and is responsible for handling client requests for service composition. A client and the associated exit node are shown in Fig. 4.3.

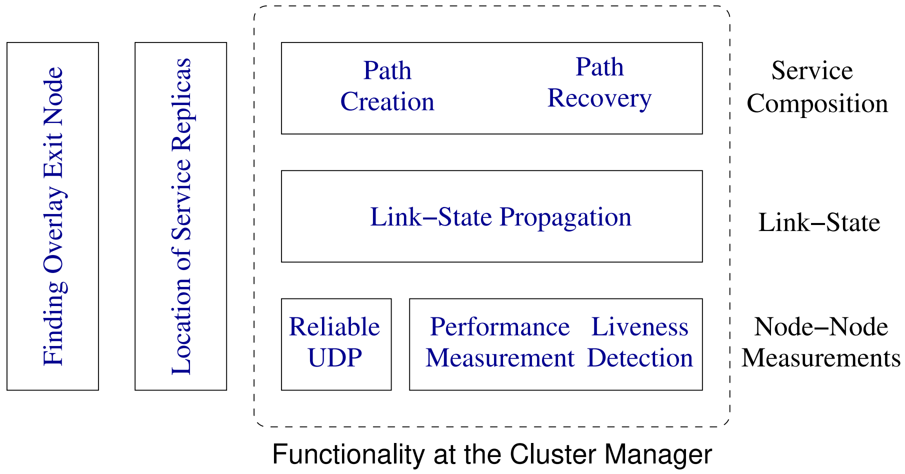

Figure 4.4: Software Architecture of the Various Functionalities

For a particular client, the choice of the exit overlay node could be made using pre-configuration, or some simple selection mechanism. Fig. 4.4 shows the various software functionalities in our architecture. The first vertical layer in Fig. 4.4 captures the functionality of **finding an exit node**.

The next functionality we separate is that of **service-location**. This is the second vertical layer in Figure 4.4. We make the important observation that this functionality is quite different from that of traditional service-discovery. Here, we just need a list of locations of service replicas – something like the list of mirrors for a web-site. Traditionally, service-discovery has combined the issues of network or service liveness/performance with that of service replica location. While this is required in a general scenario, in our operational model, we have separated the two problems, which makes the determination of service location relatively static. This can either be distributed slowly across the overlay nodes, or can simply be retrieved from a central (replicated) directory of services. The "nearness" or "liveness" of services is handled by other software functionalities. We turn to describing these now.

In each cluster, a *cluster manager (CM)* is responsible for implementing our algorithms for service-level path creation and recovery. The software architecture at the CM is also shown in Fig. 4.4. The CM implements the mechanisms for *inter-cluster*, wide-area distributed service-level path creation and recovery.

The functionality at the manager node is in three layers (Fig. 4.4). The lowest layer implements communication between adjacent nodes (service-clusters) in the overlay network. This includes liveness tracking and performance measurement. We have implemented liveness tracking as a simple periodic two-way heart-beat exchange, with a timeout to signal failure. In this chapter, we consider latency as a performance measure – our architecture also allows measurement and exchange of other metrics such as cluster load, bandwidth, or other generic metrics. Chapter 5 explores a load balancing metric in detail.

At the next layer, global information about overlay link liveness and performance is built using a link-state algorithm in the overlay network. A link-state approach gives global information about the entire overlay graph. This is used in combination with the service-location information to construct service-level paths, at the top layer.

The top layer implements the functionalities for service composition itself: initial creation, and recovery when overlay network failures are detected. The client sends the request for composition to (the cluster-manager of) its chosen exit overlay node. This CM then constructs the service-level path by choosing a particular set of service instances and paths between them in the overlay network. For this, it uses the overlay graph information built up by the link-state layer, as well as the service-location information. On choosing the service-level path, the exit node then sends signaling messages to setup the path.

The messaging at the link-state and service-composition layers are implemented on top of a UDP-based messaging layer that provides at-least-once semantics using re-transmits.

Since all the computations and control messaging relevant to composition are done at the cluster manager of each overlay node, in our discussion below, unless mentioned otherwise, we use the terms "cluster-manager" and "overlay-node" interchangeably – the cluster manager is the one at the overlay cluster node.

The algorithm used for choosing the set of service instances is a constraint-based routing algorithm. It is based on the Dijkstra's algorithm on a transformation of the overlay graph [36]. (The transformation ensures that the path chosen in the graph has the required set of service instances in the required order). We skip the details of this here since it is not relevant for our evaluation. In Chapter 5, we study how this algorithm can be used in combination with a load balancing metric to balance load across service replicas. However, in this chapter, we simply use a latency metric and choose service instances to minimize the end-to-end latency in the service-level path.

As detailed in Chapter 3, failure detection is done by the downstream node (in a data flow). This implies that we only need to use one-way heart-beats for failure detection, and this in turn means that we do not need to assume symmetry in the underlying one-way Internet paths. Failure recovery is initiated by the downstream node. When a failure is detected, there are two kinds of recovery mechanisms possible, as in MPLS [35]. We could have end-to-end path recovery or perform local-link recovery. In end-to-end path restoration, the failure information propagates downstream, to the exit-node. (Note that this handles simultaneous failure cases elegantly – the node that is the furthest downstream among all the simultaneous failures will propagate the failure information to the exit-node). The exit-node then constructs an alternate service-level path. This construction resembles the original path construction process. In local-link recovery, the failure is corrected locally, by choosing a local path to get around the failed edge. Both kinds of recovery are shown in Fig. 4.3 using dotted lines.

Finally, we make one crucial observation. We note that service-level paths have an explicit session setup phase, and there is connection-state at the intermediate nodes. The overlay network is *virtual-circuit* based, as opposed to being datagram-based. For instance, for a transcoder service, this switching state includes the input data type and source stream, and the output data type and next-hop destination information. This is also an important difference between our service-level overlay network and other overlay network approaches [20, 22, 86, 69, 79]. This means that, unlike Internet routing, failure information need not propagate to the entire network and stabilize before corrective measures are taken[1]. This is an important aspect of the system that allows quick restoration of client sessions. The time-to-recovery depends only on the time taken for the recovery messages to be processed, and not on the scale of the overlay network. (We evaluate time-to-recovery in Section 4.3).

The connection-state of a service-level path has to be maintained for the duration of a client session. We have made a design choice to have this state maintained by means of a *soft-state refresh* mechanism, and a timeout period to discard the state. This frees us from the trouble of explicit tear-down messages, and guaranteeing that these messages do traverse the path to be torn down. Such guarantees would be especially hard when tearing down an old path during path recovery – the tear-down messages might not get through the original path due to the same failure as we are trying to recover from. The connection-state is passed on from the downstream exit node, towards upstream nodes. This reflects an RSVP-style state maintenance [31]. (So far as the session-setup and maintenance is concerned, the signaling protocol we use is similar to RSVP, except that we use

[1]In the case of inter-domain Internet routing, BGP uses a path-vector protocol [70]. In [54], it is shown that the path exploration process after a route withdrawl can take O(N!) time, where N is the number of nodes in the network. Although convergence is much quicker in practice, the path exploration process is time-consuming and can take several tens of seconds to several minutes for convergence.

different information in the messages for service-level path setup, while RSVP is used for resource reservation).

4.1.4 Scale of the overlay network

In our architecture, an important issue is that of the size and extent of the overlay network. We discuss this now. We first note that the portion of the service-level path after the exit node is not "protected". That is, failures on this portion of the path are neither monitored nor recovered. Hence, ideally, each client should have an exit node "close" to it. It should be close in the sense that the client should experience roughly the same network connectivity to the rest of the Internet as its chosen exit-node. In this sense, the overlay network should span the Internet. The question then is, how many overlay nodes are required to achieve this.

As a point in comparison, we consider the Autonomous-System (AS) network in the Internet. By definition, it spans the Internet since the AS network is what constitutes the Internet. Also, by definition, each node within an AS has the same inter-domain connectivity to the rest of the Internet – just like in the definition of "close" in our case in the previous paragraph. Note that this is true even if an AS may be geographically large and disperse – all nodes within it have the same inter-domain connectivity. The AS network had about 12,000 nodes as of Dec 2001.

Another useful point of comparison for the size of the overlay topology is another Internet-wide service in operation – the Akamai content-distribution network of cache servers [2]. While this network is not an "overlay" network in that it does not do routing of user-data, it is similar to our overlay network in that it is an Internet-wide service. Here too, the goal is to span the Internet so that there is a cache server close to each client. This service had an expanse of 1000+ ISP network locations as of Oct 2001 [18] (the total number of servers was much higher, over 13,000).

As an estimate, using these two points of comparison, we can say that a few thousand nodes are probably sufficient to span the current Internet. Making a stronger claim about the exact number of overlay nodes required is an interesting research issue in itself and is out of scope of this work. However, we use the ballpark figure of a few thousand nodes (for the overlay size) for the purposes of our evaluation below.

4.1.5 Potential scaling bottlenecks and sources of overhead

Each of the layers of functionality in Fig. 4.4 has overheads. There are two different issues of scale that arise.

The first is with respect to the number of simultaneous client sessions. At the service-composition layer, the presence of connection-state per path makes quick failure recovery easier. This is because recovery is per client session, and does not depend on propagation and stabilization of the failure information across the network. However, this could have scaling implications since a large number of client sessions may have to be restored on failure of an overlay link.

The second scaling issue concerns the size of the overlay network. During path creation or restoration, finding a path through a set of intermediate service instances involves a graph computation based on the information collected by the link-state layer. This could have memory or CPU bottlenecks for a large overlay network. Further, the choice of a link-state approach for building global information could pose problems. Link-state flooding consumes network bandwidth, and this could be a potential source of bottleneck for a large network.

In our architecture, the third dimension of scaling: number of service instances, is subsumed in the scaling of the overlay network. This is because of two reasons: (a) we have separated the issue of service location from that of service instance liveness and reachability, and (b) each overlay node service cluster can be provisioned with additional service instances as demand grows, without affecting any of the algorithms for service composition in our software architecture.

Our main goal in the rest of this chapter is to identify sources of scaling bottlenecks, quantify the various overheads, and determine how quickly we can effect service-level path recovery. We now turn to describing our evaluation testbed to study these overheads.

4.2 Experimental Testbed

In the evaluation of a large scale system, the choice of experimental methodology is very important. Simulation testbeds are not appropriate and large-scale wide-area distributed testbeds are not viable for this. We explain our choice of a network-emulation platform in Section 4.2.1. We then quantify the system bottlenecks that we encounter in our emulation testbed in Section 4.2.2. We later use this set of measurements to verify that in our experiments, we are within the operational limits of the testbed.

4.2.1 Network Emulation Platform

Our design involves a system that stretches across the wide-area Internet. A design study of the system is challenging for several reasons. We have the issue of scale in the dimensions of number of clients and expanse of the system. A controlled design study also requires repeatability of experiments. We first consider the choice of a simulation-based evaluation. This would involve a particular modeling of Internet dynamics, and a dummy implementation of our algorithms for path choice and recovery. Such an approach would have the advantage of repeatability, and would give some quick results.

However, this methodology has serious problems. Firstly, simulation test-beds are known to have severe scaling problems both in terms of the number of nodes, and more importantly, in the number of client sessions. Although the theoretical limit of simulators like ns-2 [14] is a few 100 nodes, most simulation studies we are aware of use not more than a dozen nodes. This is because simulations tend to be orders of magnitude slower than real experiments. A more serious problem is with scaling in the dimension of the number of simultaneous client sessions. Typical simulation experiments usually handle only a few 10's of client sessions at best. This is woefully inadequate for our purposes since we wish to study scaling with 100's/1,000's of simultaneous client sessions. Furthermore, while simulations are good for studying protocol behavior, they are ill-suited for identifying processing bottlenecks.

At the other extreme is the choice of a wide-area testbed using a real deployment. While this would definitely be a more authentic study, it has serious problems too. It is very time-consuming and/or expensive to create and maintain such a testbed. Furthermore, the experiments are highly non-repeatable, making it ineffective for a controlled design study to evaluate alternative algorithms head-to-head, or to identify system bottlenecks. This also makes initial testing and debugging of code quite difficult.

Hence we choose to use a *network-emulation platform* for our experiments. Here, the idea is to run a *real* implementation of the algorithms and mechanisms, on multiple machines in a cluster environment, but simply *emulate* the wide-area network characteristics between the machines. The emulation is done at a layer below the application. Such an approach would have the advantage of repeatability, as in simulations, but would be closer to reality than a simulation implementation. Very importantly, it would allow us to have all the nodes under our control (unlike a real testbed), and would allow us to scale in the dimension of the number of simultaneous client sessions (unlike a simulation testbed).

The opportunity for such an emulation-based platform is provided by the Millennium cluster of workstations [7].

Our emulation platform was inspired by NistNET [8]. NistNET is a kernel-level implementation of an emulation platform. Incoming network packets are intercepted by a kernel module and are delayed before being passed to the application, or dropped, based on a set of rules. The rule-set is configurable at the user-level through a set of ioctl commands. We do not use NistNET for two reasons. Firstly, we faced operational difficulties in introducing NistNET, a kernel-level emulator, in a production cluster shared by other researchers. Secondly, our own implementation of an emulator gave us better control over varying network parameters for our evaluation.

We have implemented our version of packet modifier at the user-level using raw IP sockets. This requires super-user privileges. Again, since the cluster was in use by other researchers, and was centrally managed, we could not have such a packet-modification at every node. We had to have all traffic pass through a single node that runs such a packet modifier – we call this machine the *emulator*. This testbed setting is shown in Figure 4.5(a).

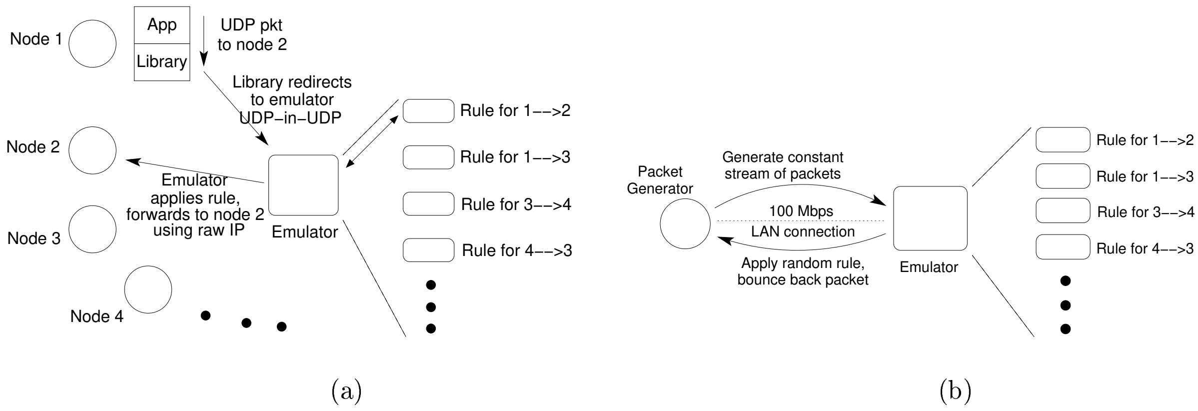

Figure 4.5: Emulator setup

Note that this emulation cluster is quite different from the service-clusters in our architecture. In fact, each node in our emulation setup represents a cluster manager of a service-cluster/overlay-node in our architecture, and runs the software shown in Figure 4.4. The algorithms are real in implementation – the code is finally re-linked with a library that redirects packets via the emulator node. The emulator, besides acting as a router, has rules for capturing the behavior of each "overlay link" between pairs of overlay nodes (Figure 4.5(a)). In our architecture, the actual application data traffic does not pass through the cluster manager. And hence in our emulation too, we only capture the control-traffic between the cluster managers.

We have modeled delay/latency behavior between overlay nodes, as well as the frequency and duration of failures of the overlay link. The actual settings for these packet handling rules, and the choice of the overlay topology itself, are presented in Section 4.3.1. We now identify the bottlenecks in this testbed setting.

4.2.2 Bottlenecks in the Emulation Platform

The emulator node in our testbed is a potential source of bottleneck since all traffic passes through it. This is especially the case since our implementation of packet handling rules is at the user level. We now quantify the limits of the emulator by performing experiments to stress it. We later use this to check that in our actual experiments, we do not exceed the capacity of the emulator.

Each emulation node in our testbed is a 500MHz Pentium-III machine with up to 3GB memory, and a 500KB cache. Each is a 2-way, or 4-way multi-processor, and runs Linux 2.4. The emulator is setup on a Pentium-4 1500MHz machine with 256MB memory, and 256KB cache, running Linux 2.4.2-2. The emulator software is started up with a configuration for a 40-node overlay topology (topology generation and packet handling rules explained in Section 4.3.1). There is a single machine that generates traffic at a given rate. On receiving a packet, the emulator chooses a random rule to fire. The rule may indicate a packet drop, or a particular delay value for the packet. In the latter case, after the delay, the emulator returns the packet to the traffic generator. This is depicted in Figure 4.5(b).

We collect logs at the traffic generator as well as the emulator. To prevent the explosion of the log for high traffic rates, we only log the behavior of a fraction of the packets. Such packets to be logged are chosen at random by the traffic generator, and "flagged" so that the emulator also knows to log its behavior.

We vary two parameters: the rate at which packets are sent by the traffic generator, and the packet size. For a single experimental run, the packet size is fixed. We consider two metrics that represent deviations from expected behavior: (1) the delay introduced by the emulator, in addition to that dictated by our latency model, and (2) the percentage of packets that are dropped by the emulator, in addition to that dictated by our overlay link failure model (both the models are described in Section 4.3.1).

Table 4.1 gives the first metric (average additional delay, in milliseconds), and Table 4.2 gives the second metric (% packets missed by the emulator). Both are presented for different packet rates (along columns), and different packet sizes (along rows). The packet rates are given in packets/sec, and the packet sizes in bytes. The numbers in parentheses in the first table give the standard-error across all packets in the experimental run.

In Table 4.1 and Table 4.2, the scaling limits of the emulator are reached in both dimensions – at large packet sizes due to the memory copies for each packet, and at high packet rates, due to the kernel/user crossing for each packet. For instance, for packet sizes of 500 bytes, the scaling

	10,000/sec	15,000/sec	20,000/sec	25,000/sec
250B	0.003 ms (0.12 ms)	0.001 ms (0.11 ms)	0.002 ms (0.07 ms)	1.26 ms (0.56 ms)
500B	0.002 ms (0.10 ms)	0.005 ms (0.09 ms)	0.307 ms (0.51 ms)	38.0 ms (1.38 ms)
800B	0.020 ms (0.20 ms)	55.66 ms (4.0 ms)	55.74 ms (2.32 ms)	55.86 ms (2.12 ms)
1100B	0.520 ms (0.76 ms)	74.48 ms (2.84 ms)	74.27 ms (3.36 ms)	74.60 ms (2.56 ms)
1400B	92.95 ms (5.26 ms)	92.98 ms (3.47 ms)	93.23 ms (3.36 ms)	93.21 ms (3.42 ms)

Table 4.1: Emulator performance: additional latency in emulator

	10,000/sec	15,000/sec	20,000/sec	25,000/sec
250B	0.000%	0.020%	0.005%	23.9%
500B	0.010%	0.020%	0.185%	20.4%
800B	0.86%	8.72%	29.24%	44.1%
1100B	1.63%	36.14%	49.75%	64.71%
1400B	36.36%	50.65%	65.48%	68.95%

Table 4.2: Emulator performance: % packets lost by the emulator

limit is reached at 20,000 packets/second. This translates to 500 $bytes/packet \times 8\ bits/byte \times 20,000\ packets/second = 80\ Mbps$. This is close to the limit of the 100 Mbps ethernet. For a packet size of 250 bytes, the scaling limit is reached at 25,000 packets/second. Since the data rate in this case is only 250 $bytes/packet \times 8\ bits/byte \times 25,000\ packets/second = 50\ Mbps$, the likely bottleneck here is the per-packet processing at the emulator. This is understandable given that the implementation is at the user-level.

However, we note that the emulator performs quite well for up to a packet rate of 20,000 packets/second, for packet sizes below 500 bytes. (These limits are at the emulator node, and are independent of the number of other nodes). We shall refer back to these numbers later to verify that in our experiments, we do not exceed these limits of operation of the emulator.

4.3 Evaluation

In this section, we turn to the evaluation of our system. We seek to understand the scaling behavior of the system and quantify overheads as summarized in Section 4.1.5. In our set of experiments, we consider several metrics: (a) the time to *recovery* of client path sessions, after failure detection, (b) the additional control overhead due to spurious path restorations, and (c) other memory, CPU, and network overheads in our software architecture.

We study client session recovery time as a function of the number of client sessions (load) at each CM. We analyze the two different recovery algorithms presented in Section 4.1.3. The set of experiments presented in this section are as follows.

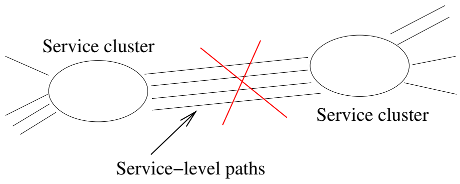

Figure 4.6: The issue of scaling during failure recovery

- In Sec. 4.3.2, we consider end-to-end path recovery and study its scaling behavior. We study scaling in the dimension of the number of clients. The essence of the scaling issue here is the presence of per-session switching state at the CMs. A whole set of client sessions may have to be restored on failure of an overlay link. This is depicted in Figure 4.6. The restoration involves changing the switching state for each session by setting up an alternate path. We examine this scaling limit, and in turn, the limit it imposes on the number of client sessions that can be handled by a CM (its load).

- In Sec. 4.3.3, we compare local recovery with end-to-end recovery. Local recovery has the advantage that recovery time can be lower since the recovery messages are sent locally. However, it has the disadvantage of not using global information and this might result in higher-cost paths. We examine the nature of this trade-off.

- For the above set of experiments, we use realistic modeling of Internet delay, but use controlled link failures. We then use the trace data presented in Chapter 3 to model Internet path failure behavior, and we study the time to path recovery under realistic Internet failure patterns in Sec. 4.3.4. This allows us to examine spurious path restorations. We quantify the rate of occurrence of timeouts and the overhead it imposes.

- Finally, we look at other sources of overhead in our system in Sec. 4.3.5. We quantify the overheads imposed by the heart-beat messages, link-state floods, and graph computations. We examine if these impose any scaling limits on the system.

We use our emulation testbed for the design study and evaluation. Before presenting our experiments, we explain two important parameter settings in this subsection: the overlay topology, and the nature of performance variation of the links in the overlay network.

4.3.1 Parameter settings for the experiments

The Overlay Network Topology

While we envision a full-fledged deployment of our architecture to constitute a few thousand overlay nodes (Sec. 4.1.4), we first wish to study system behavior with a smaller number of nodes. This is also a limitation imposed by our emulation testbed which has a maximum of a hundred machines to act as overlay nodes. (This is the number of available machines in the Millennium cluster). However, we study scaling in the dimension of the number of client sessions with this setup.

We use the following procedure to generate an overlay network. We first generate an underlying *physical network* with a Transit-Stub topology. This graph has a total of 6,510 nodes, and 20,649 edges. We choose this network size since it is "big enough" to generate overlay networks of sizes up to a few hundred nodes, by randomly picking a subset of nodes as overlay nodes (details described below). This network size is also "small enough" to accommodate memory and CPU intensive algorithms like the all-pair-shortest-path graph algorithm on the commodity hardware we had at our disposal (we use these algorithms for the generation of the overlay network). The 6,510-node physical network topology is generated using the GT-ITM package [84] (with 14 transit ASes, each with 15 nodes, 10 stub-ASes per transit-node, and 3 nodes per stub-AS; these set of parameters give a hierarchical topology).

We select a random subset of N nodes from the physical network to generate an N-node overlay topology (N is a much smaller number than 6,510). Next, we examine pairs of overlay nodes in the order of their closeness (computed using the all-pair-shortest-path algorithm) and decide to form overlay links between these. Overlay links are thus equivalent to *physical paths*. In this process, we impose the constraint that no physical link is shared by two overlay links. Although this could theoretically result in a disconnected overlay topology, for the graph that we used, the final overlay network was connected.

Overlay Network Parameters

To study our mechanisms for service-level path creation, adaptation, and recovery, we vary two network parameters: latency, and occurrence of failures (packet drops are modeled simply as short failures). We use these two parameters to capture the nature of overlay links in our emulations. Each rule at the emulator involves these two parameters.

Latency Variation: To model this, we use results from a study of round-trip-time (RTT) behavior on the Internet [17]. We make use of two results: (1) Significant changes (defined as over 10 ms) in average RTT, measured over 1 minute intervals occur only once in about 52 min. This value of 52 min is averaged over all host-pairs. (2) The average run length of RTT, within a jitter of 10 ms, is 110 seconds across all host-pairs. The first result says that sustained changes in RTT occur slowly, and the second result says that the jitter value is quite small for periods of the order of 1-2 minutes.

We use these as follows. The costs of edges of the physical network are as generated by the GT-ITM package. For the overlay links, the cost is simply the addition of the costs of the physical path edges between the overlay nodes. This cost is however, only relative. We normalize this by setting the maximum overlay link cost of *100 ms* – this is the one-way cost. We thus get a base-value for the latency in an overlay link. Given a base-value L for the latency, we vary the latency between L and $2L$. Such a variation of overlay link cost gives a maximum one-way latency of $2 \times 100 = 200\ ms$, and a max RTT of up to $2 \times 200 = 400\ ms$. This is a reasonable choice since overlay links are likely to be formed between "close-by" overlay nodes – they are unlikely to be separated by an RTT of over 400 ms. We impose the constraint that significant sustained changes happen once in an "epoch" of length 52 min (using result (1)). Also, to have some variability, we set a value of 15 min for this epoch for 10% of the overlay links, and 100 min for another 10% (the rest 80% have the value of 52 min). Within an epoch of RTT value, 1 min averages are varied within 10 ms (in accordance with (1)). And within a minute, jitter is within 10 ms (in accordance with (2)).

In our modeling of latency variation, we do not include occasional, isolated RTT spikes that do happen [17]. Instead, we model RTT spikes also as loss-periods/failures, which is worse than RTT spikes. (Although the study we have used is somewhat old, it is extensive. Also, our own UDP-based experiments in Chapter 3 agree in spirit with observation (2) above – in our experiments, we observe that outage periods lasting beyond 1-2 sec are very rare).

Occurrence of failures: For the initial set of experiments, we fail graph links in a controlled fashion. We then used a trace-based emulation of network failures. We use the traces from Chapter 3 for this.

4.3.2 Time to path recovery: end-to-end recovery

In this subsection, we study the system behavior with an increasing number of simultaneous client sessions, while using an end-to-end recovery mechanism for failed service-level paths. We capture our metric of time-to-recovery of client sessions as a function of the number of client sessions for which an overlay node is the exit node (and hence its CM is responsible for path creation and recovery for that session). In the rest of the discussion, we refer to the number of client sessions for which an overlay node is an exit node as the *load L* on it (or equivalently, the load on its CM).

In this set of experiments, we first use a 20-node overlay network (with 54 edges) generated as described earlier, and study scaling in the dimension of the number of clients. We later consider the effect of increasing the overlay size. There are a total of ten different services, "s0" through "s9", each with two replicas in the overlay network. Having two replicas ensures an alternate server for failure recovery, and having ten different kinds of services with two replicas each ensures that the overlay network is uniformly covered with services. The replicas are placed at random locations in the overlay. Each client path request involves two different randomly chosen services from among the ten. (Note that although each path has only two logical services, the path could stretch across many more overlay nodes, via the no-op services).

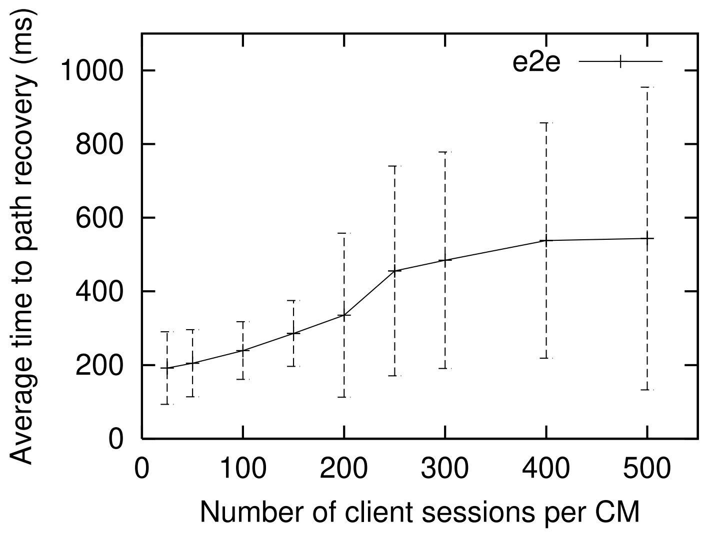

Figure 4.7: Time to recovery vs. Load

Across the runs, we vary the load L from an initial value of 25 paths per CM, and increase it gradually to examine the scaling behavior. We have equal load at all the 20 CMs. For a given load, we first establish all the paths (total #paths = #paths terminating at a CM $\times$ 20 CMs). We then deterministically fail the link in the overlay network with the *maximum* number of client sessions traversing it. This is the worst case in a single-link failure. We conclude the experiment shortly after all the failed paths have been recovered (a few seconds). We then compute the time to recovery, averaged over all the paths that failed and were recovered. Fig. 4.7 shows this average metric plotted against the load as we defined above. The error bars indicate the standard deviation.

There are several things we note about the plot. Firstly, the average time to recovery remains low, below 600 ms even for a load of up to 500 paths per cluster manager. This time-to-recovery is the time taken for signaling messages to setup the alternate path, after failure detection. Secondly, the average time-to-recovery increases only slowly as the load increases. This suggests that the system has not reached its saturation point yet. That is, even at higher load, queuing delays associated with processing the distributed recovery messages are minimal. The third observation we make is that the variance of the time-to-recovery across all failed paths is large at high load. To explain this, we plot another graph.

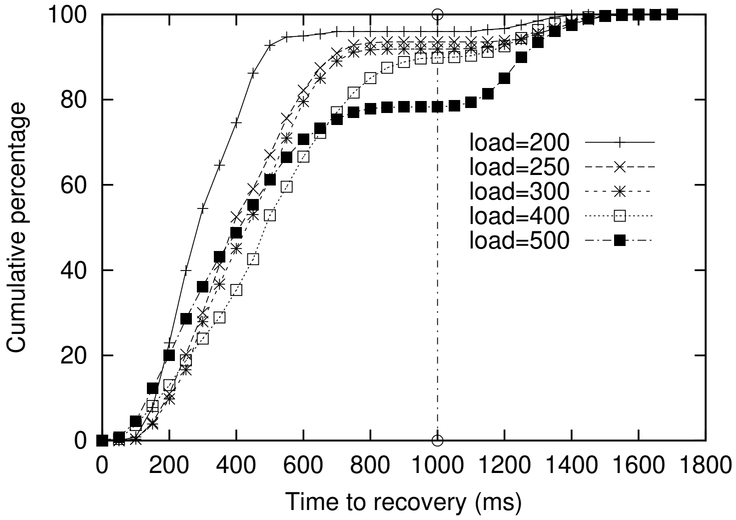

Figure 4.8: CDF of time-to-recovery for different values of load

Fig. 4.8 shows the CDF of the time-to-recovery of all the failed paths. Different plots are shown for different values of the load. We see that the majority of the paths recover well within 1 sec, and a small fraction of the paths take over 1 sec to recover (notice the flat region in the CDF). This is due to the following reason. The path recovery control messages are transmitted using the reliable UDP messaging layer of Fig. 4.4. This layer implements a re-transmit after 1 sec, if there is no reply to the first packet[2]. Such a re-transmit occurs for the path recovery control messages since the first control message is lost, at higher load. A certain fraction of the paths being recovered thus experience significantly higher recovery time than others. This explains the high variance at high load, in Fig. 4.7.

There are two reasons why packet losses can occur: (1) excess load in processing the path recovery messages at the CMs, or (2) bottleneck at the emulator in our setup. Note that we have not yet modeled packet-losses/outages on the overlay links (this is considered in Section 4.3.4). Also, the control packet losses could not have been caused due to the deterministically failed link, since

[2]We use a value of 1 sec for the first re-transmit, 1.5 sec for the second re-transmit, 2 sec for all further re-transmits.

Load L	#pkts lost by CMs	#pkts lost by emul.	Max rate at emul. (pkts/s)
200	0	99	20,640
250	0	163	19,240
300	0	201	19,630
400	0	578	21,200
500	0	753	21,590

Table 4.3: Detecting the bottleneck

our algorithm does not send any recovery messages on the failed link itself. Case (1) implies that we have a bottleneck in our software architecture, while case (2) would mean that the emulator setup is being stressed. To check this, we instrument the emulator to: (a) count the number of packets it sent and received, and (b) measure the packet rate it saw, in 100 ms windows. The CMs also keep track of the number of packets they send and receive. Using (a), we compute the number of control packets lost at the CM, and the number of control packets lost in the emulator setup. We use (b) to check against the emulator limits given in Table 4.2 of Section 4.2.

In Table 4.3, we tabulate these values for different loads. We notice that there are no packet losses at any of the CMs, meaning that the bottleneck is not in the message processing at these nodes. However, the emulator node (or the local area network in-between) loses a small number of packets, and this number increases with the load in the system. The table also gives the maximum rate seen by the emulator in 100 ms windows. Referring back to Table 4.2, we see that the emulator setup is close to its limits in these experiments, in terms of the packet rate. (The sizes of all control packets were within 300 bytes). Note that for every packet lost by the emulator, a client session recovery could experience a control message re-transmit, and thus a recovery time higher than 1.0 sec.

We thus conclude with certainty from the above experiments that the system can handle at least 200 paths/CM easily. Also, since *no* packets are lost by the CMs due to processing bottlenecks (column 1 of the Table 4.2) even at higher loads, we can say with reasonable certainty that the scaling limits of the CMs have not been reached even at loads of 400-500 paths/CM. This is also corroborated by the fact that the average time-to-recovery increases only slowly with increasing load – if saturation point had been reached, we would have expected to see a steep increase in the plot at this saturation point.

What this scaling limit means is that we would have to provision additional CMs to handle clients beyond this limit. (Although we have not stated this in our architecture description, we could have multiple CMs per cluster, with each CM handling a different set of client sessions).

Our cluster manager machines are Pentium III 500MHz quad-processor machines. During our experiments, since the cluster was in production use, we were not able to get fully unloaded machines, but always used the least loaded set of machines. The number of 400-500 simultaneous paths per cluster manager is a reasonable number, since we are dealing with heavy-weight application services such as video transcoders, text-to-speech converters in our examples given earlier. These services could be compute-intensive (e.g., text-to-speech), data-transfer-intensive (e.g., multicast to unicast protocol transformation agent for video), or both (e.g., video transcoder). For comparison, the text-to-speech service we implemented in [67] could support only about 15 simultaneous client sessions on hardware similar to those running our CMs. This means that in deploying a service cluster, the amount of provisioning required for cluster manager functionality would be small in comparison to that required for actual services such as the text-to-speech engine. Also, note that a cluster can have multiple CMs dealing with different sets of client path sessions – the system can be provisioned with more cluster managers to support a larger number of simultaneous client sessions.

We make another observation. We have used latency as a metric for path creation, and in the above experiments, failed the overlay link with the *maximum* number of client paths traversing it. This represents a worst-case scenario. This is because, as is well known, a metric such as latency is very poor in distributing load across the network. In fact, in our experiments above, we observed that the load across the overlay nodes was highly skewed. The system can be expected to scale even better if a load balancing metric such as cluster-load is used. We have implemented such a load balancing metric, described in Chapter 5.

In the above experiments, we have not considered scaling along the dimension of the number

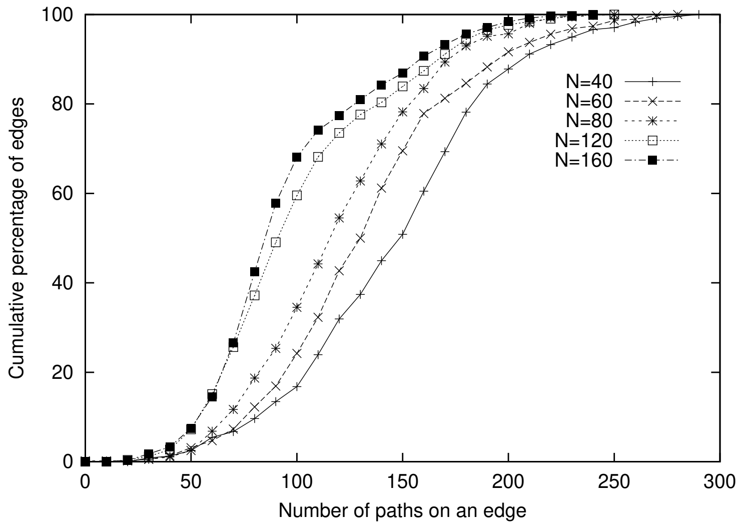

Figure 4.9: CDF of edge loads, for various overlay sizes

of overlay nodes. However, intuitively, if we grow the overlay network size, and correspondingly also increase the number of service replicas, the load on the overlay links should remain the same irrespective of the size of the network. In fact, we do observe this experimentally. We generate overlay topologies of various sizes, as explained in Sec. 4.3.1. We choose a number of service replicas in the overlay proportional to its size. We place these replicas at random locations in the overlay. We setup a number of client sessions, and then measure the load on each overlay link. We plot a CDF of this edge load across all the edges in the overlay, for different overlay sizes. Fig. 4.9 shows this set of plots. We see that as the network size grows, the load distribution across the various edges does not change much. In fact, with greater connectivity for the larger networks, the edge load only evens out. This is suggested by the fact that the CDF becomes more vertical at the middle with increasing overlay size. Even the maximum edge load does not change with growing overlay size. These observations mean that a link failure in a larger network, with a proportionally larger number of clients, is no worse than in a smaller network. That is, scaling with respect to the number of clients does not worsen with increasing overlay size.

4.3.3 Time to path recovery: local recovery

We now examine the alternate method of *local* recovery where the failed edge is replaced by a local path. This recovery mechanism has the advantage over end-to-end recovery that since the signaling messages are local, the recovery time can be lower. However, since the path is being fixed locally, we might lose out on global optimization. That is, the resultant path after local recovery might have a higher cost than if end-to-end recovery had been used. We look at the nature of this trade-off now.

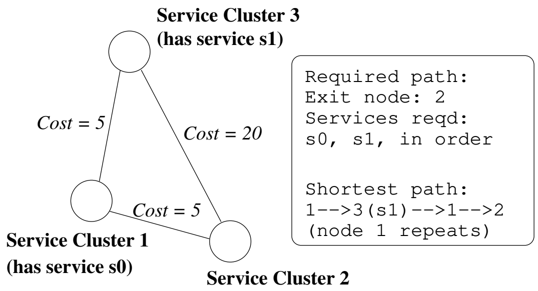

Figure 4.10: Node repetition: an example

Like in our earlier set of experiments, we have a set of runs with varying load; in each run, we create paths before-hand, and then fail the overlay link with the maximum number of paths going through it. Apart from the trade-off mentioned above, there is a further issue with local recovery. Since paths are constrained to pass through nodes with services, they may not be simple graph paths: they may have repeated occurrences of nodes or edges in them. An example is shown in Fig. 4.10. Since local recovery hides the recovery information from the rest of the nodes in the path, handling race conditions in distributed messaging, when there are multiple occurrences of nodes in the original path, becomes difficult. For this reason, we fall back on end-to-end recovery when the original path has repeated occurrences of nodes.

Hence in each run, we use local recovery for client sessions whose original paths do not have repeated nodes, and end-to-end recovery for other client sessions. In each run, there were a significant fraction (at least 25%) of client sessions in each category – it was not the case that one kind of recovery was applied for most client sessions in any run. This has the side effect of making

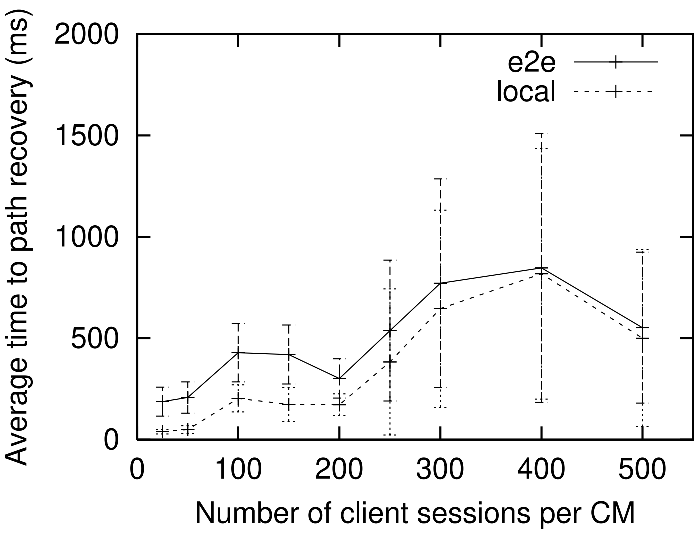

Figure 4.11: Local vs. E2E recovery (time-to-recovery)

our comparison simpler, since we can compare the average time-to-recovery of paths, under either algorithm, in the same run. The two plots in Fig. 4.11 and Fig. 4.12 illustrate the trade-off between the two algorithms. The first graph shows the average time-to-recovery as a function of the load, much as in Fig. 4.7. The second graph shows the other metric: the ratio of the cost of the recovery path, to the cost of the original path, as a function of the load. (Recall that the path cost in our case the end-to-end latency).

In the first graph (Fig. 4.11), we note that the time-to-recovery has low values, around 700 ms, as earlier. Also, the variance in the time-to-recovery goes up with load, as in Fig. 4.7. The small non-uniformity in the plot is understandable given the magnitude of the variance. Another point we note is that local recovery has consistently lower average recovery time, as expected. Although it has lower time-to-recovery, we note that the difference is very low in absolute terms – within 200-300 ms. As our discussion in Chapter 3 showed, these small differences are dwarfed by the time to failure *detection* in Internet paths – about 1.8 sec.

The second graph (Fig. 4.12) shows the flip side of local recovery – it results in paths that are costlier than with end-to-end recovery. Here, the difference between local and end-to-end recovery are significant. Local recovery results in paths that are 20-40% costlier than the original

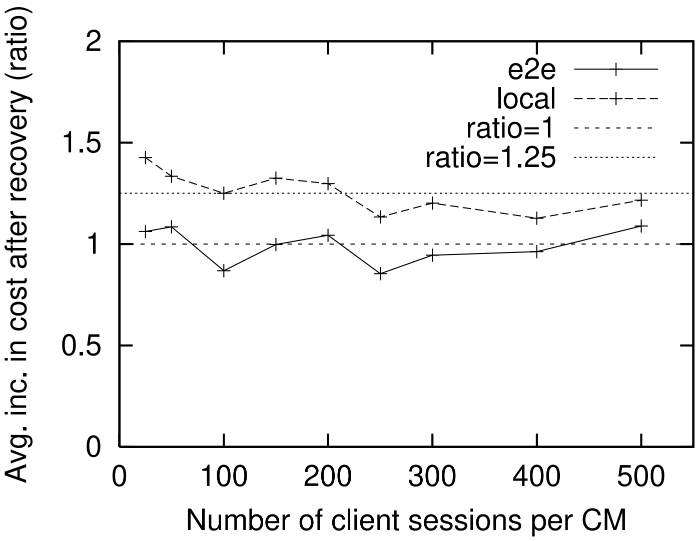

Figure 4.12: Local vs. E2E recovery (path cost)

path, due to the additional re-route in the middle of the original path. On the other hand, end-to-end recovery causes a maximum extra cost of 10% over the original path, and in many cases actually improves the path cost over the original path. Improvement in path cost over the original path is due to the following reason. The latency metric along overlay links is variable, as explained in Sec. 4.3.1. Hence the original min-cost path is no more the min-cost path after a while – at the time of path recovery. Hence, when an alternate end-to-end path is setup, it can incur a lower cost than the original path. While these differences of 10-30% one way or another may not greatly affect the performance of the client path when using the latency metric, it is significant if we use a graph metric such as load on the cluster node.

4.3.4 Performance under Internet failure behavior

In this experiment, we wish to study two things: (a) the extent of spurious path restorations under Internet outage patterns, and (b) the performance of our recovery messaging under Internet packet losses as given by our traces in the prior section. Given the set of CDFs of outage durations in the earlier section, we fail links in our overlay with a particular probability, for a particular duration, according to the distribution as in Fig. 3.3. For an overlay link in the testbed, we choose one of the

18 distributions at random. We have a fixed timeout of 1.8 sec to detect failures between a pair of overlay nodes. We now run the same experiment, with the 20-node graph, with a load of 300 paths per cluster manager (total number of paths in the system = $20 \times 300 = 6000$). We use only the end-to-end recovery algorithm for this run. We let the system run for a period of 15 min.

During the run, across all the 54 edges in the graph, there are 162 outages that last 1 sec or more, of which 32 outages last 1.8 sec or more, and 7 last for 20 sec or more. There are 11,079 end-to-end recovery attempts triggered. This represents an average of about two recoveries per client path session during the experimental run. 10,974 (99.05%) of these recovery attempts were successful.

For a number of the shorter outages, the outage time itself is comparable to the recovery time. Such short outages are, in some sense, false-positives that trigger spurious path restorations. Ideally, these should not have triggered any recovery – but this happens due to our aggressive timeout mechanism to detect failures quickly. To quantify the fraction of spurious path restorations in our experimental run, we count the number of recovery attempts that were a result of a failure lasting less than 3 sec.

We find that, of the 11,079 recovery attempts, 6,557 (59.18%) are caused by such short outages. This figure of about 60% for the fraction of spurious restorations triggered merits some discussion. We first note that even if a recovery attempt is spurious, application data is not lost any more than during normal Internet performance, without our recovery algorithms. This is because the original path is torn down only after the new path has been established. The only overhead of a spurious recovery attempt is in the control messages introduced by our service composition layer. The control overhead itself is minimal, and can easily be handled with little additional provisioning in terms of cluster managers, as shown in Sec. 4.3.2. In absolute terms, spurious path restorations and failures themselves occur infrequently. The average rate of occurrence of failures per link in our experimental run is: $\#outages\ over\ 1.8\ sec/\#links/15\ min\ =\ 32/(54 \times 2)/0.25\ =\ 1.2/hour/link$. The rate of occurrence of spurious restorations is even lower since only a fraction of the outages represent spurious failure detections. Hence spurious restorations are a small price to pay for the benefits of quick failure detection with an aggressive timeout.

An important aspect of path restorations (including spurious ones) is that of system *stability*. If the absolute rate of occurrence of path restorations is high in the system, instability could result. That is, paths could be switched repeatedly, with cascading or alternating failures due to overload in portions of the overlay network. In our experiment above, we did not observe any such instability. In retrospect, the reason for this is simple – our system can easily handle loads of 300 paths/CM (which is what we had in the experiment above), and there are no processing bottlenecks that drive the system to an unstable state.

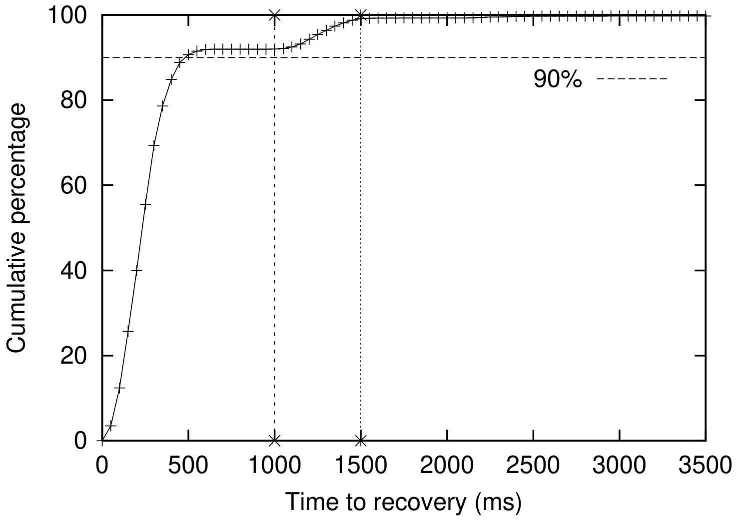

Figure 4.13: Performance under realistic failures

Fig. 4.13 shows the CDF of the time-to-recovery of all the paths. Note the flat region in the CDF, as in Fig. 4.8. This represents a re-transmit of a control message during path recovery. Such re-transmits are due to the Internet packet losses we have modeled in this experiment. The plot indicates that over 90% of the recoveries are completed within 1 sec. This represents the recovery time under realistic packet loss as modeled by our outage periods. Such a quick restoration represents orders of magnitude better performance than Internet path recovery that takes several tens of seconds to minutes [54].

4.3.5 Other sources of overhead

So far we have focused on the path recovery algorithm component of our architecture. The other pieces are (1) the peer-peer heart-beat and measurements, (2) the link-state propagation, and (3) the path creation algorithm itself. The first consumes minimal resources: the heart-beat is sent every 300 ms in our implementation. And peer-peer latency measurements are done once every 2 sec. The bandwidth consumed by these is minuscule.

The second, link-state propagation, is performed whenever there is a change in the link-status (dead/live), or when there is a significant change in the latency over the link. Apart from this, we also have a soft-state link-state propagation every 60 sec to handle dynamic graph partitions. Given the nature of latency variation as described earlier, sudden large changes in latency are rare. So most link-state floods are sent over the network due to link failures or restorations. In the experiment we described in the previous subsection, 150 link-state floods happen over the entire run of the experiment lasting 15 min, notifying nodes of a link failure or link recovery. Given that a link-state flood means a single message over each link in the graph, there are only 150 messages per link due to these floods over the entire run. This is also minimal. We expect this number to increase linearly as the number of edges in the graph increases. This is not too bad however, since we do not stipulate a complete graph for the overlay network as in [20]. In fact, it is ideal if the overlay network maps onto the underlying physical network closely. That is, it is good if multiple overlay links do not share underlying physical links. Hence we can expect the number of overlay edges to be a small multiple of the number of overlay nodes (network topologies generated by GT-ITM for example, have only 4-5 times the number of edges as the number of nodes). This means that even for large overlay graphs with a few thousand nodes, the number of overlay edges is likely to be of the same order of magnitude. As a rough estimate, consider a 2,000-node overlay graph with 10,000 edges. Using the data from our traces in Chapter 3, we have timeouts about once an hour on each overlay link. The rate of link-state floods due to the up/down events would hence be $2 \times 10,000\ edges \times 1\ timeout/hour \times 1/3600\ hours/sec \simeq 6\ floods/sec$. That is, there would only be six messages/sec on each edge of the network, due to the link-state floods. This would consume minuscule amount of bandwidth.

Another possible source of overhead is the graph computation involved during path creation and path recovery. The complexity of Dijkstra's algorithm is $E \times log(N)$, where E is the number of edges and N is the number of nodes in the graph. In Section 4.1.3 we mentioned that the algorithm is applied on a transformation of the overlay graph. In turns out that this transformation does not affect the algorithm complexity. In our implementation, this algorithm performs quite well. We performed micro-benchmark studies (not an emulation run) of this algorithm alone, with a 6,510-node overlay network, with 20,649 edges. On the configuration of our cluster machines, the computation takes about 50 ms, and only about 3MB of memory. This figure of 50 ms could be significant overhead if this computation is done for every path creation or recovery. However, we perform an optimization that we term *path caching*. We run the algorithm, and store the resulting "tree" structure for requests for path creation/recovery in the near future. We store one such tree for every kind of service-level path (not every client path session). We update this tree only when the graph state changes – i.e., only 150 times, once for each link-state update, during our experimental run in the previous sub-section. Since we do not run this algorithm for every path creation/recovery, this is not a source of bottleneck.

4.3.6 Summary of results

In summary, our results show that failure recovery can be performed in our overlay network of service clusters, within 1 sec for over 90% of client sessions (Sec. 4.3.4). Our trace-data, and the experiments using those show that failure detection can be quite aggressive, with a timeout as low as 1.8 sec, with an infrequent occurrence of spurious path restorations – about once an hour in our experiments. Hence, overall, paths can recover from outages within about $1 + 1.8 = 2.8$ seconds. This would be of tremendous use to applications such as video streaming – without our mechanisms for recovery, client sessions could experience outages that last for several minutes [54]. This figure of 2.8 seconds is definitely good enough for real-time, but non-interactive applications, which usually buffer about 5-10 sec of data (e.g., text-to-speech, video-on-demand). For interactive applications such as voice-over-IP or interactive video, this may not be perfect, but would provide significantly better end-user experience than without our recovery mechanisms.

Our data shows that there is no bottleneck with the control message processing involved during path recovery, so far as we have been able to scale our emulation testbed. We explored the use of local recovery – while this results in quicker recovery under low load, the local nature of the recovery could lead to sub-optimal path metric for the recovered path.

4.4 Chapter Summary

In this chapter, we have presented our architecture for wide-area service composition. We build our recovery mechanisms on top of the failure detection mechanism described in Chapter 3. The architecture is based on a service-level overlay network of service clusters. The use of clusters helps us separate the issue of process- and machine-level failures from wide-area network path failures. Quick recovery is achieved in the overlay network since the overlay is virtual-circuit based and recovery does not depend on propagation and stabilization of failure information across the network. Our emulation-based design study has shown that the bottleneck in the system is the fact that a large number of client sessions have to be recovered simultaneously on an overlay link failure. This imposes a limit on the number of simultaneous client sessions per CM. Scaling beyond this limit requires additional provisioning in each cluster. However, the additional provisioning required for this is minimal since we are dealing with heavy-weight service components such as text-to-speech engines or multimedia transcoders.

In this chapter, we have focused on the design of the architecture. The evaluation and design study have dealt with the recovery mechanism, the related metric of time-to-recovery, and the associated scaling bottlenecks. In the next chapter we shift focus to a design study of algorithms for choice of service instances during service-level path creation as well as recovery.

Chapter 5

Load Balancing Issues in Wide-Area Service Composition

An important goal of our architecture for wide-area service composition is that of *performance*. By this we mean an appropriate choice of service instances and network paths for the construction of a service-level path. There could be several replicas of the different services at the various service locations. We need to choose lightly loaded service instances and ensure load balancing among the replicas. We also need to ensure adequate network performance for the data flow in the service-level path.

The overlay network in our architecture and the link-state based graph state propagation provide the context for exchange of performance information: service instance load information as well as network path performance information. In this chapter, we focus on the mechanism for load balancing among the various service replicas.

Load balancing in any distributed system consists of several components including: (a) a feedback loop between the point where load is experienced and the point where decisions are made, and (b) a mechanism to use the feedback to drive future decisions of where to place load. These have to be designed to prevent load oscillations and to provide stable behavior under a variety of conditions.

Although the problem of web-server selection has been researched in the past [33, 83, 23, 76, 53, 77, 45] in the context of an Internet-wide distributed system, there are several aspects of service composition that make our work novel. First, we have to choose a *set* of service instances to form a service-level path, and not just a single web-mirror. Second, composed client sessions could involve real-time media and the session could last for several minutes to hours. We consider load balancing in the presence of failures *during* a session. These considerations lead to an altogether different architecture and set of mechanisms for load balancing.

We introduce a metric for choosing a set of service instances for a composed client session: the *least-inverse-available-capacity (LIAC)* metric. This is used to assign costs to edges in a graph with service replicas at different nodes; the least cost path in this graph is chosen as the service-level path for the client. We first try a mechanism for load information dissemination based on periodic updates from the service replicas. Though this does well, we find that it causes load oscillations. We then introduce a *piggybacking* mechanism to update load information via the service-level path setup messages. This does not update load globally, but only along the service-level path, and has little additional overhead. Despite the fact that piggybacking updates load only along the service-level path, we find that it can achieve very good load balancing and can effectively reduce oscillations.

Piggybacking achieves good load balancing across replicas, but the LIAC metric often chooses far away service instances. This results in longer service-level paths, and hence in larger end-to-end latency for the client session. We introduce an additional factor in our LIAC load balancing metric – this achieves a good trade-off between length of service-level path and load balancing between service replicas. We find that this load balancing metric performs well under a variety of scenarios, including failure recovery of service-level paths *during* a client session.

The rest of the chapter is organized as follows. In the next section, we briefly present the problem statement of load balancing in the context of our architecture. We then present our mechanisms for load balancing in Section 5.2. We discuss the load balancing metric as well as the piggybacking mechanism for updating load information. Section 5.3 presents experiments with our load balancing mechanism under a variety of scenarios. Section 5.4 presents concluding discussions.

5.1 Problem Statement

Service composition uses component services to enable new applications. The reusability of the independent components for different compositions gives flexibility. We envision a scenario where independent service providers deploy and manage their service instances at multiple locations on the Internet. Other third-party portal providers compose these for end-users. While there are several challenges in the context of service composition, in this chapter we focus on load-balancing across the replicas of services placed at different Internet locations.

Recall that our architecture is based on a service platform consisting of several service clusters deployed at different Internet locations (see Fig. 1.3). Individual service providers deploy their services at these service clusters. The service clusters form an overlay network that enables service composition. The service network is an overlay in the sense that it is constructed on top of the IP layer. Each service cluster is thus an overlay node (we use these terms interchangeably in the rest of the paper). Service-level paths are constructed by choosing a set of required service instances and forming a *path* in the overlay network.

The different kinds of component services at the service clusters could be content sources (e.g., the video-on-demand server) or could be data transformation/personalization agents (e.g., the text-to-speech engine). In addition to these, we also have "no-op" services that can be instantiated at each service cluster on demand. An example is shown in Fig. 1.3. These no-op services do not change the data in any way and only provide connectivity. This enables composition of services that are not necessarily in adjacent clusters of the overlay network.

When a service-level path stretches across service clusters, the Internet path in-between could span multiple domains in the wide-area network. An important concern is that of availability of the service-level path. In Chapter 3, we described how network path failures can be detected using periodic heart-beats between service clusters. We recover failed service-level paths by choosing an alternate service-level path for the client session.

The construction of the original service-level path as well as alternate paths for recovery is done at a service cluster that we term the *exit overlay node* for the particular client session. Each client chooses an overlay node that is "close" to it for all service composition. Data traverses through the overlay nodes along the service-level path and exits the overlay network at the "exit" node (see Fig. 1.3).

All communication and messaging is done at the *Cluster Manager (CM)* machine of a service cluster. The CM is responsible for running the algorithms for selecting the specific set of service instances needed to setup a composed client session. Once the set of instances have been chosen, the exit-node CM sends control messages along the service-level path to instantiate the services as well as the no-op services as required.

Since in our architecture, each overlay node is a service cluster, we focus on load balancing *across* service clusters. There has been past research in load balancing across machines within a cluster [40, 63] and we leverage on this.

Thus, in abstract terms, we have a graph that represents an overlay network. Each node in the graph has a set of services. We assume that the set of locations for each kind of service is known globally (this is similar to the knowledge of the set of mirrors for a web-site). Paths in the graph have to be chosen to satisfy "constraints" – a set of services have to be traversed in a particular order. Client requests can come in at any graph node (each graph node may be an exit node for a particular set of clients on the Internet). Each service when instantiated at a graph node, adds a particular value to the load at the node for the duration of the client session. Each graph node has a particular capacity with respect to the amount of load it can handle. This capacity would in practice depend on the provisioning at the service cluster. We assume that each machine within the cluster can be used to instantiate any of the services present at the cluster. That is, there is no per-service provisioning at the service clusters. (For instance, if a service cluster has five machines and three kinds of services "s0", "s1", and "s3", any of the five machines can be used to instantiate any of the three services).

In the context of programmable networks, a graph algorithm for constructing paths with

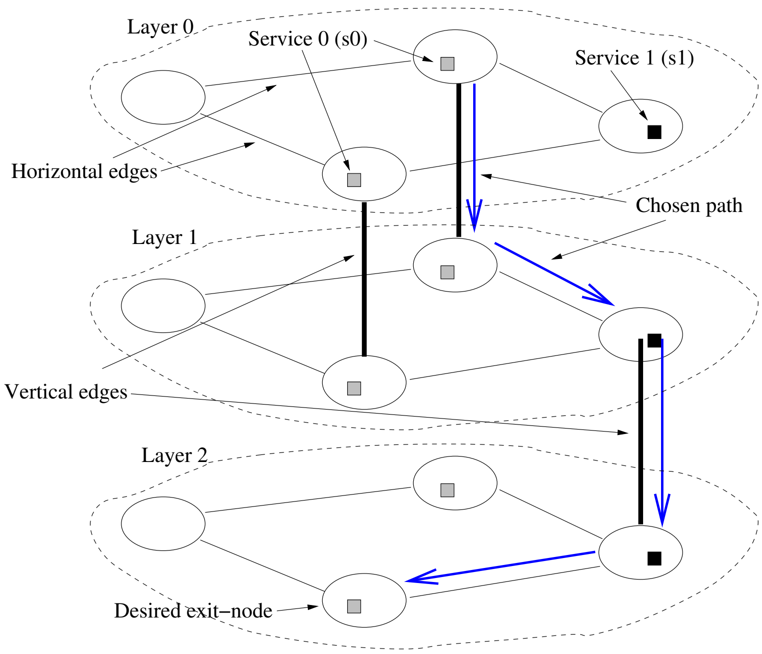

Figure 5.1: Graph modification for service composition

intermediate processing sites in presented in [36]. This algorithm applies the well known Dijkstra's algorithm [38] for least-cost computation in a transformed graph. While [36] presents a generic algorithm, it does not say what metrics/costs should be used for the graph edges. In our work, we use the generic algorithm and graph transformation presented in [36], but we focus on how the graph edge metrics/costs can be set, as well as its interaction with load information dissemination. We now briefly summarize the graph transformation in [36]. Given the original network graph and the location of the different services, and a client request involving k intermediate services, the graph modification consists of replicating the graph $k+1$ times. Vertical edges are added at nodes where the required services are present. A simple example is shown in Fig. 5.1 where the client requests for the composition of two services "s0" and "s1". Vertical edges are added between the first two layers at the nodes where the service "s0" is present; and vertical edges are added between the bottom two layers at the nodes where the service "s1" is present. Any path from the top layer to the exit-node at the bottom layer will thus pass through a node with service "s0" and then a node with service "s1".

5.2 Load Balancing

We now turn to a discussion of the issue of load balancing. Balancing load is important to ensure overall good performance. Periods or regions of overload can result in poor end-to-end performance of the client session. Or, in case admission control is used, it could lead to rejection of client requests. The essential issues with respect to load balancing in a distributed system include: (a) the design of an appropriate feedback loop to convey information about load-increase/decrease from where it happens to where decisions are made (e.g., from server to the client, or between nodes in a network), and (b) the mechanism to use this feedback to drive future decisions.

We consider two main factors in the design of the load balancing mechanism: (i) load variation across replicas as well as load oscillations over time, and (ii) the length of the service-level path in the overlay graph – this has to be minimized since we do not want to choose service instances away from the client's exit node.

We now present the design of the feedback loop for load balancing, and its interaction with the load balancing metric. In Sec. 5.2.1, we present the *least-inverse-available-capacity (LIAC)* metric for choosing a set of service instances for a service-level path. We study its interaction with a periodic link-state-based dissemination of load information. In Sec. 5.2.2, we introduce a piggybacking mechanism for updating load information along a service-level path as it is being setup. We address the issue of overall latency and length of the service-level path in Sec. 5.2.3. We address this by introducing a "no-op" factor in the LIAC metric.

5.2.1 Load balancing: basic mechanism

Our mechanism for load balancing consists of two components: (a) mechanism for load information dissemination, and (b) mechanism to use this load information. To disseminate load information, we use a simple link-state-based[1] approach where each node periodically floods its load information to the rest of the network. We need a way to set the costs of the edges in the transformed graph (Fig. 5.1), based on the information about the different nodes' load. This is

[1] "Link"-state is a misnomer here since what we are flooding is really "node"-state; i.e., its load.

so that the Dijkstra graph computation can then be applied to arrive at the required service-level path. A metric that is simply the addition of the current loads at the nodes along a service-level path is unlikely to perform well. A consideration of the total capacity and the currently available capacity at a service cluster (graph node) is important. (The available capacity at a service cluster is the difference between the maximum load it can handle and its current load level). We are thus motivated to think in terms of an inverse function of the available capacity at a node. We borrow intuition from research in QoS literature. A metric that is known to work well for choosing network paths with requisite bandwidth guarantees is the least-distance metric [58]:

$$PathCost \;=\; \Sigma_{link\ \epsilon\ path} \frac{1}{AvailableBandwidth_{link}} \tag{5.1}$$

The intuition behind this metric is that the cost of using a particular link is inversely proportional to the bandwidth available on it currently. That is, the closer to capacity a link is, the less likely that it will be used. The simulations in [58] show that this metric can achieve low call blocking rate. This means that the metric is good at distributing different clients' data across the links of the network. In our scenario, we are concerned not just with bandwidth balancing on links, but more importantly with balancing load across cluster overlay nodes. This is important since server load often has a greater effect on the client session "quality".

Since service instances are central to service composition, and since we are concerned with server load balancing, we are motivated to try a metric that is derived from the inverse of the available capacity at an overlay node:

$$PathCost \;=\; \Sigma_{S\ \epsilon\ path} \frac{1}{MaxLoad_S - CurrLoad_S} \tag{5.2}$$

Here, S represents a node on which a particular service is meant to be instantiated. $MaxLoad$ and $CurrLoad$ represent the maximum load a particular service cluster overlay node can take, and its current value, respectively. This metric is implemented by assigning a cost to each of the vertical edges in the transformed graph (Fig. 5.1); this cost is the inverse of the available capacity at the graph node corresponding to the vertical edge. We apply the Dijkstra's algorithm on the transformed graph with this cost assignment to get a desired service-level path. (We choose the minimum-cost path from a node of the top layer to the desired exit node at the bottom layer). Intuitively, this metric favors overlay nodes that have the maximum difference between $MaxLoad$ and $CurrLoad$, just as Equation 5.1 favors network links with the maximum available bandwidth. We term the metric in Equation 5.2 as the *least-inverse-available-capacity (LIAC)* metric.

We are interested in the performance of this metric, and its interaction with the link-state-based load information dissemination. We study this using an emulation platform. We now present the emulation setup followed by our experimental results.

The Emulation setup: We use the same emulation setup based on the Millennium cluster of workstations [7] as described in the previous chapter. We have a real implementation of the algorithms and emulate wide-area latency. Each node in the Millennium cluster emulates the functionality of one (or in some of our scaling studies, more than one) overlay node. We emulate only the cluster manager functionality since we are interested only in the behavior of the system as represented by the exchange of signaling messages between the cluster managers. We generate the overlay network as earlier. The latencies assigned to the various links, and the variation of this latency are also as in Chapter 4.

Setup for the study of the LIAC metric: For our study of the behavior of the LIAC metric, we use an emulation setup with a 40-node overlay network, with 119 overlay links. This suffices for the purpose of studying the LIAC metric now; we consider larger overlay networks in Section 5.3. The emulation is set to run on 40 different nodes of the Millennium testbed. We have 10 different services in the network: "s0"-"s9". Each overlay node implements exactly one kind of service (apart from the special "no-op" service) and there are 4 replicas for each kind of service. Having four different replicas allows us to study the load variation across these replicas (we consider different numbers of service replicas in Section 5.3). And having ten different kinds of services ensures that each overlay node has a service replica.

We setup client sessions at an overall rate of 20 requests/sec, with each client path session lasting for a duration varying uniformly between 70 and 90 sec. (Intuitively, a faster arrival rate of clients would only increase the load variation. The choice of client session duration is driven by the fact that we are interested in long-lasting sessions. The nature of our observations are independent of this parameter – we verified this in other experiments). The experiment lasts for 400 sec (actually a little longer, including startup time for the software), with 20 $requests/sec \times 400\ sec = 8,000$ paths being setup totally. The duration of 400 seconds allows several sets of client sessions lasting 70-90 seconds to be setup and torn down. This allows us to observe the long-term behavior of the load variation, and examine load variation over time. The exit-node for each client session setup is chosen at random from among the 40 nodes. Each client session requests for a composition of two randomly chosen services. We fix the link-state update period to be 60 sec. We stipulate that the load addition due to an instance of each of the 10 kinds of services is the same: a value of 1. We fix the $MaxLoad$ for each overlay node to be 2500[2].

While we have chosen this set of parameters for showing our results, the nature of the results remain the same with other parameter settings as well. We hope to convince the reader of the same as we present the range of scenarios in this section and the next.

Results: Table 5.1 shows the number of client paths which used each of the four replicas for services "s0"-"s4". (This number is the total for the run of the experiment, and not for any particular instant). We see that this metric does reasonably well in terms of load balancing across the service replicas, but for some shortcomings. In the case of all the five services shown in the table, there was one replica that was loaded consistently less than the others. We explain this below. A plot of the time-variation of the load across the different replicas is more informative. Such a plot is shown in Fig. 5.2 for the replicas of the service "s0" (the plot for the other services look similar). The four replicas are placed at graph nodes with IDs 8, 19, 26, and 38. (Graph nodes are numbered 1-40; SCID stands for service cluster ID – recall that each graph node represents a service cluster in our architecture). The y-axis represents the instantaneous load, measured at each 10-second interval, and the x-axis represents time.

We observe large variations over time in the load at each of the four replicas of the service.

[2]We experimented with other values of MaxLoad; the qualitative nature of the observations remain the same.

Replica number	s0	s1	s2	s3	s4
1	461	440	140	238	493
2	462	170	496	438	226
3	176	499	448	452	494
4	458	470	579	467	369

Table 5.1: Load distribution across server replicas

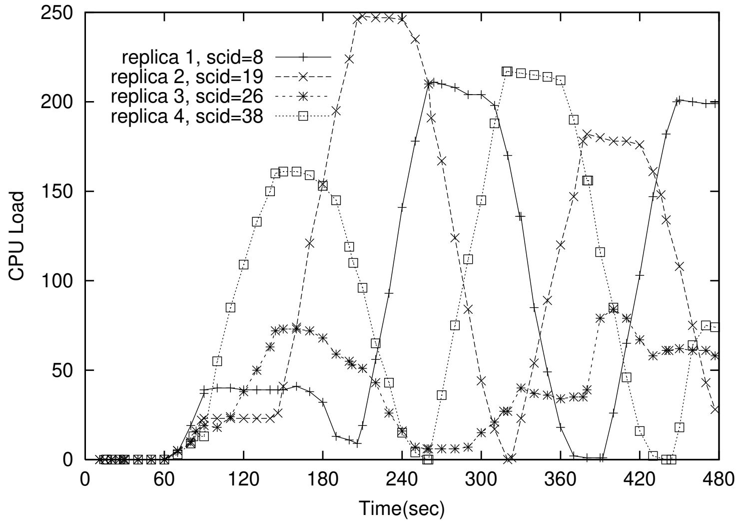

Figure 5.2: Load variation with the load-balancing metric

There are periods when the load at a service replica increases steadily, and periods when the load decreases steadily. We also observe that the duration of these periods of load increase/decrease is about 60 seconds – the same as the link-state update period. To confirm this correlation, we re-run the experiment with a link-state update period of 30 seconds. This plot is shown in Fig. 5.3. Here again, we observe that there can be periods as long as 30 seconds during which the load at a service replica keeps increasing, or keeps falling.

The fact that the constant load increase/fall duration matches the link-state update period offers an explanation for the variation. Although we have requests equally distributed across all the overlay nodes, load variation happens in-between link-state updates. If the load increases during a cycle, the link-state update causes the load to drop during the next cycle, and vice versa. In short, the feedback loop for carrying load information is not quick enough to prevent load oscillations.

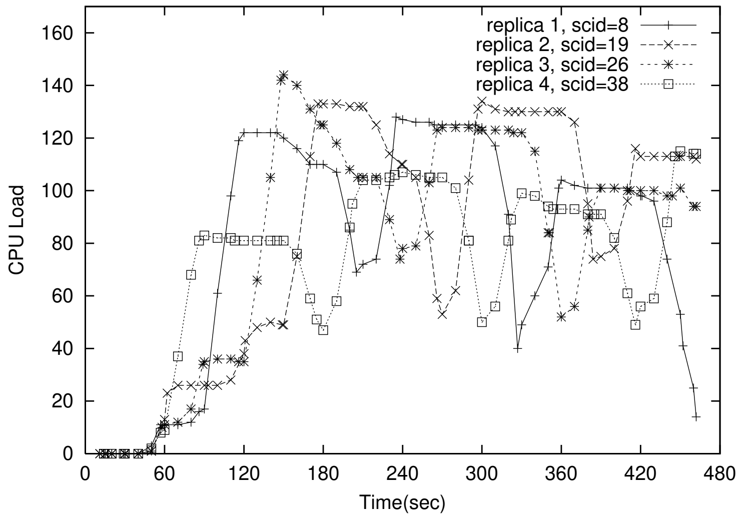

Figure 5.3: Effect of lower link-state update period

The load variation also offers an explanation for the behavior observed in Table 5.1. The phases of load variation for three of the replicas happen to be such that, most of the time, there is one of the three that (seemingly) has a very low load in comparison to the fourth replica (this fourth replica with low load in Table 5.1 is replica #3 at node 26). This fourth replica thus get used much lesser than the other three.

5.2.2 Piggybacking

Load variations are not good because if the system is operating close to capacity, parts of it will be driven to overload during periods of time. Even if the system is operating well within its overall capacity, if a part of it gets a lot of client requests all of a sudden, such load variation, due to lack of a good feedback mechanism, could cause that part of the system to be driven to overload.

We now turn to the mechanism for reducing load oscillations. Two possible approaches are to reduce the link-state update period, or to have on-demand link-state updates. In the on-demand approach, we flood the network when there is "substantial" change in load information since the time of the previous flood. We reject both of these approaches for different reasons. Having frequent link-state floods increases the overhead in the system, especially for larger networks. On the other hand, having on-demand link-state updates is not desirable due to the following reason. If and when the system load increases rapidly, on-demand updates would generate a lot of link-state updates. That is, we would be adding more to the system load especially when it is experiencing overload. This could potentially lead to instability.

Instead of these two approaches, we introduce a mechanism where we leverage the service-level path setup messages to piggyback load information. The path setup messages traverse downstream to upstream and an acknowledgment is generated upstream to downstream. We piggyback load information in either direction. Each node in the path reads the piggybacked load information, and adds its own load information to the message. Note that this mechanism would update load information only along a service-level path, and not along the entire graph (a link-state flood updates load along the entire graph).

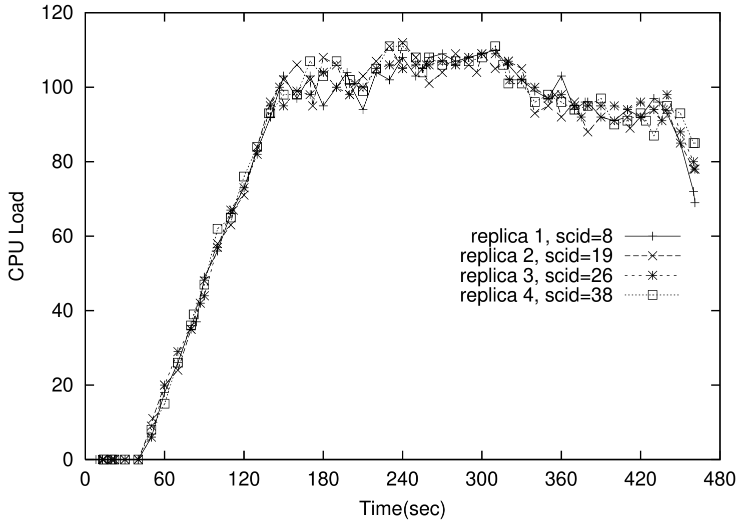

Figure 5.4: Effect of piggybacking load information

We experiment the performance of this piggybacked load update mechanism with the same emulation setup as in the previous sub-section. Fig. 5.4 shows the load variation across the same four service replicas as earlier. We observe that the load across the four replicas follow the same trend at all times throughout the experiment. The flat region in the graph starts when we have as many paths timing out as there are new paths being created – the overall system load level is constant at this point. This was not apparent in the previous plots due to the oscillations.

Piggybacking load information only along the portions of the network on which client paths are setup is thus able to achieve near-perfect load balancing. Piggybacking has several nice properties. First, load updates are as frequent as client path setups, without much additional cost. Hence we can expect periods of overload (when there are a lot of path setup requests) to be handled gracefully. In comparison, frequent or on-demand flooding of load information would have had a lot of overhead. The second nice property about piggybacking is with respect to its handling of "load information discrepancies" – that is, wrong information about load at a particular server replica. Such discrepancies happen in a distributed system since no node can have perfect global information at any instant. Wrong information could be of two kinds: underestimate of load, or overestimate. In a system trying to do load balancing, underestimates are especially bad since this could cause the portion of the system whose load is underestimated to be driven to overload (an underestimate means that the server actually has high load, but everyone thinks it has low load). With piggybacking, the behavior with underestimate is good since the moment a client request is made to the server whose load is underestimated, the feedback from the load information piggybacked on the path setup messages would immediately correct the underestimate. That is, underestimates are inherently short-lived with the use of piggybacking.

The effects of overestimate are not as bad since it would simply mean that the replica would remain unused. Piggybacking will not help here since the replica remains unused. However, after a link-state update corrects the load information discrepancy, since the load at that replica was small to begin with, it would be used. And since it would be used, the exit nodes using it for client requests would get piggybacked load information about the replica.

5.2.3 No-op factor

The combination of piggybacking-based load updates with the LIAC metric performs well so far as load balancing is concerned. However, it has a bad effect not apparent from the results presented so far. We observe that the path length in terms of the number of hops for the service-level path in the overlay graph is too large. (This path length includes the “no-op” services in-between the instantiated services). Fig. 5.5 shows the CDF of the path length of all the 8,000 paths that were setup in the experimental run from the previous sub-section. The plot compares the case where we used a minimum-latency (ML) metric for path selection, with the case where we used the LIAC metric. The ML-metric works simply by assigning the overlay link latency as the metric for path selection.

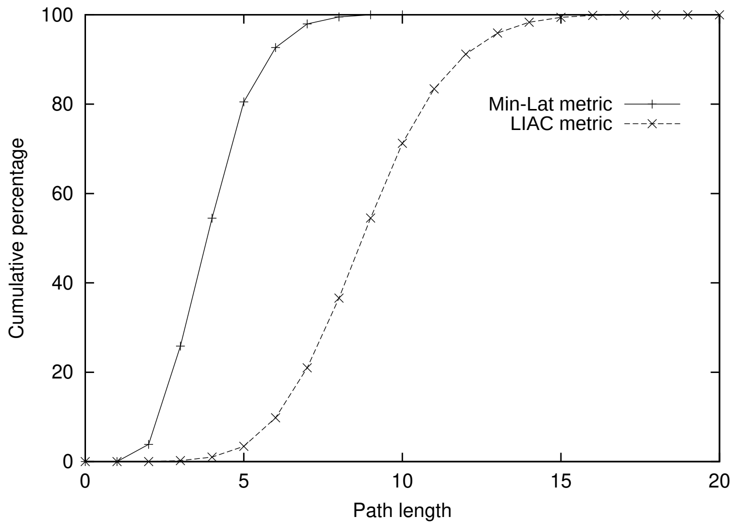

Figure 5.5: CDF of path lengths: comparison

The load-balancing algorithm performs poorly in terms of path length since it tries to optimize on the load balancing and has no factor to discourage the choice of very long paths. Hence even if a far-away service instance has slightly smaller load, it is chosen over a nearer service instance. Higher path length has several bad effects including wasted network resources (since data travels over a larger portion of the network), higher end-to-end latency in the client session, as well as greater probability of experiencing outages.

The minimum-latency metric assigns costs to the horizontal edges of the transformed graph (Fig. 5.1), and these costs correspond to the overlay link latency. (It turns out that in the overlay graph we generate, although latency and hop-count do not have a perfect correspondence, they have a high degree of correlation). In contrast, our LIAC metric assigns costs only to the vertical edges of the transformed graph.

Ideally, we would like to achieve good load balancing, while at the same time not lose out on path length. However, we note that there is no easy way of combining the minimum-latency metric with the LIAC metric since one represents latency and the other represents the inverse of the available capacity. (In the language of physics, these have different "dimensions").

The reason why the LIAC metric ends up with long paths is that it assigns no cost on the horizontal edges of the transformed graph. We now introduce a factor to account for horizontal hops as well. For each horizontal edge, we assign a cost proportional to the inverse available capacity of the node "downstream" of the edge (downstream with respect to the direction of the service-level path towards the client – this usually represents the direction of data flow towards the client). The metric is thus:

$$\begin{aligned} PathCost &= \Sigma_{S\ \epsilon\ path} \frac{1}{MaxLoad_S - CurrLoad_S} + \\ & \Sigma_{(D,U)\ \epsilon\ path} \frac{\alpha}{MaxLoad_D - CurrLoad_D} \end{aligned} \tag{5.3}$$

Here, (D, U) represents an edge on the service-level path (a horizontal edge in the transformed graph), from an upstream node U to a downstream node D. Since this metric is meant to discourage large path lengths, that is, the use of unnecessary no-op services, we term this the *least-inverse-available-capacity* metric with the *no-op factor (LIAC-NF).*

An important feature of the LIAC-NF metric in Equation 5.3 is the parameter α, which is a fraction less than 1. The intuition behind this is that we do not want to give as much weightage to reducing path length, as to balancing load between replicas. The parameter α can potentially be tuned to give more weight to optimizing path length versus giving weight to load balancing. If α is 0, this metric is the same as the LIAC metric and there is no weightage to reducing path length. (It is important that the system behavior is not particularly dependent on the value of α – we study this in more detail in a Sec. 5.3.2).

Fig. 5.6 shows the effect of using the LIAC-NF metric, with an emulation run similar to the previous ones. It compares the CDF of the path lengths of the 8,000 paths that were setup. The comparison is again with a case where we have the minimum-latency metric for choosing the service instances. This plot uses a value of $\alpha = 0.1$. We see that the path lengths are comparable and in many cases even lesser than the minimum-latency algorithm. (Recall that the minimum-latency metric need not achieve the minimum number of hops since the correlation between hop-count and latency is not perfect).

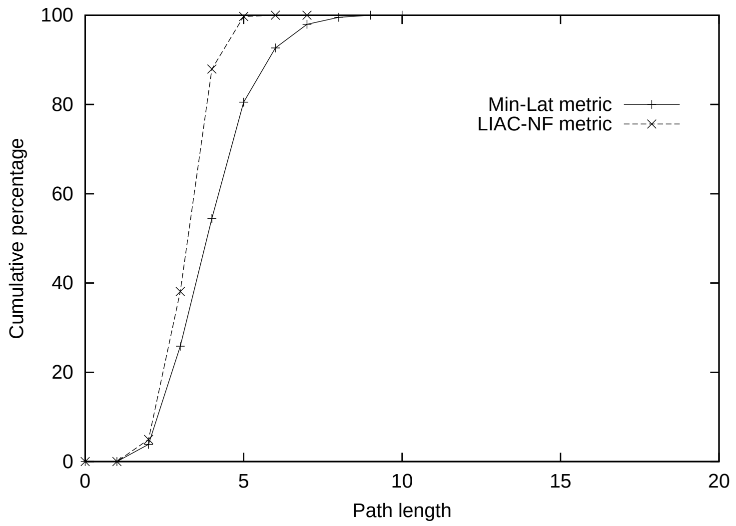

Figure 5.6: Comparison of path length CDFs, with $\alpha = 0.1$

While the LIAC-NF metric does well in terms of path length, we also wish to ensure that it does well in terms of load balancing. Fig. 5.7 shows the load variation for this case with the use of the metric in Equation 5.3. We use a link-state update period of 60 seconds again. We see that the load variation is still very less in comparison to Fig. 5.2, where we had no piggybacking. We observe more variations than in Fig. 5.4 – the case where we used the LIAC metric, which is the same as the LIAC-NF metric with $\alpha = 0$. However, these variations are small.

5.3 Behavior under other scenarios

In this section, we study the performance of the LIAC-NF metric and the piggybacking mechanism under a variety of scenarios. In particular, we consider: (a) uneven load distribution,

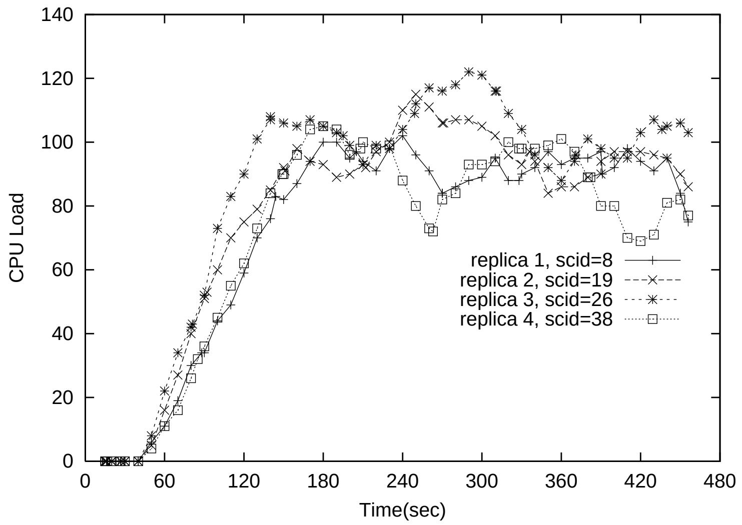

Figure 5.7: Load variation with piggybacking, with no-op factor $\alpha = 0.1$

where a portion of the network is constantly loaded more than the rest of the network (Sec. 5.3.1), (b) effect of varying α as well as the number of service replicas in the network (Sec. 5.3.2), (c) effect of increasing the size of the network (Sec. 5.3.3), and finally (d) the behavior of the system when there is single/double link failure and a large number of service-level paths are simultaneously recovered (Sec. 5.3.4 & 5.3.5).

In all these experiments, we use the LIAC-NF metric, and incorporate the piggybacking mechanism, in addition to the periodic link-state update. The link-state update period is fixed at 60 seconds. Unless mentioned otherwise, we have ten kinds of services in the network: "s0"-"s9", and results are plotted for the replicas of the service "s0" (with the results for the other services being similar). Also, unless mentioned otherwise, we use a path setup rate of 20/sec and setup a total of 8,000 paths.

5.3.1 Effect of uneven load

So far we have not considered the effect of uneven load distribution in terms of path creation requests coming into the different overlay nodes. We now introduce uneven load by having 80% of the path creation requests coming into 20% of the overlay nodes. Fig. 5.8 shows the load variation in this scenario. We set $\alpha = 0.1$. We see that although the incoming request load is uneven, the LIAC-NF metric and the piggybacking mechanism are able to achieve good load balancing across the replicas.

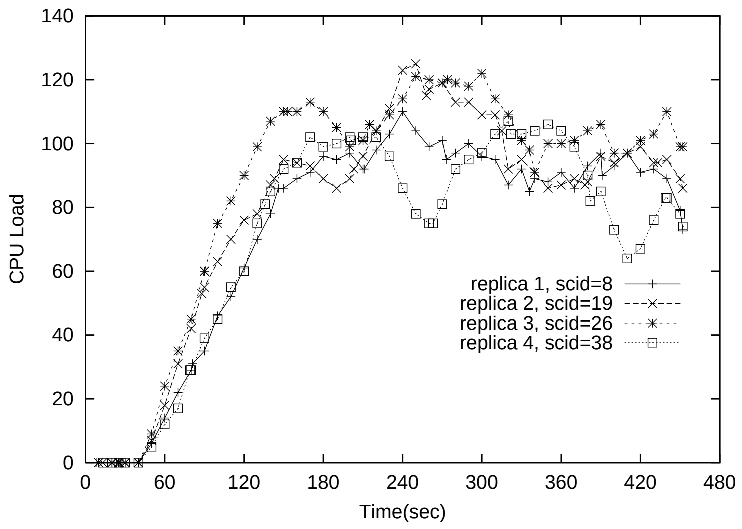

Figure 5.8: Load variation with uneven incoming load

5.3.2 Varying α

The parameter α determines the trade-off between path length and load-balancing. It is desirable that the system performs well in terms of both measures (path length, and load-balancing) for a range of values of α, since then tuning this parameter would not be an issue. We wish to study the effect of varying α. Alongside, we also wish to see the effect of varying the number of services. This is because, intuitively, the path length is also determined by the availability of close-by service instances, and in turn by the number of service replicas in the network.

For these set of experiments, we represent the results in a more compact form than in the previous plots. For the path length measure, instead of showing the CDF of the lengths of all the paths setup, we simply show the average path length. And instead of showing the load variation over time, we simply show the ratio of the maximum loaded node and the minimum loaded node. We call this load-balancing metric as the *max-min-ratio (MMR)*. Since this might be an extreme measure, we also show the ratio of the next-to-maximum loaded node and the next-to-minimum loaded node (note that this might be less than 1 if we have only two service replicas, and will be exactly 1, if we have three service replicas). We term this metric the *next-to-max-min-ratio (N-MMR)*. MMR as well as N-MMR are measured at an instant, and not using the max/min values of load over the duration of the experiment. The ideal values for these ratios is 1, when all replicas have the same load. We show these two ratios as measured at the end of the setup of 8,000 paths, for the case of service "s0".

Fig. 5.9 shows the variation of the average path length for different values of the number of service replicas. Each line represents a different value of α. We see that except for the case where $\alpha = 0$, the path length is comparable for all other values. The path length reduction by increasing the value of α by an order of magnitude, from 0.01 to 0.1 is very small.

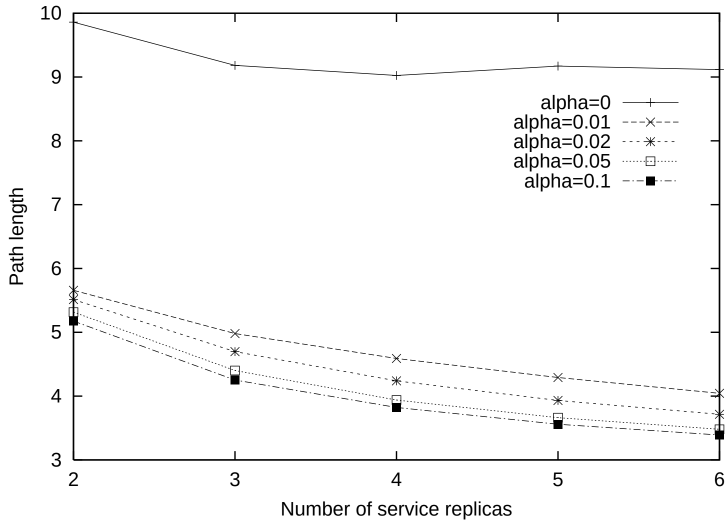

Figure 5.9: Path length variation with α

Fig. 5.10 shows the variation of MMR with the number of service replicas, for different values of α, and Fig. 5.11 shows similar plots for N-MMR. We see that for a range of the number of service replicas, and for different values of α, the LIAC-NF metric performs well in combination with the piggybacking mechanism.

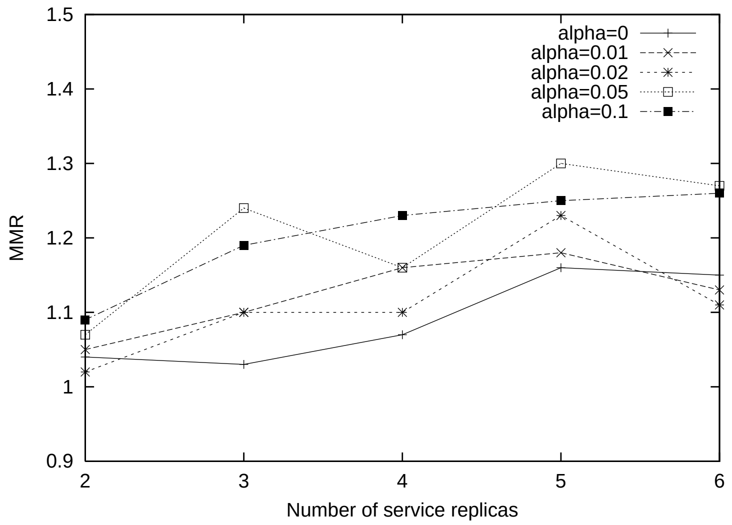

Figure 5.10: Max-min-ratio (MMR) for different values of α

5.3.3 Scaling the number of overlay nodes

As the scale of the overlay network grows, the feedback loop for the load-balancing algorithm has more delay. We now show the effect of a larger overlay network. We generate overlay graphs as described earlier, with number of nodes varying from 40 to 160. In these experiments, we have 20 different kinds of services ("s0"-"s19"), and there are enough replicas so that each node had exactly one kind of service. Thus in the 40-node configuration, each service had two replicas, and in the 160-node case, each service had 8 replicas. The value of α was fixed at 0.02 for all these experiments. The rate of client path request arrival as well as the number of paths created are proportional to the number of overlay nodes. For the 40-node network, the rate of request arrival was 80/sec and the number of paths 10,000. For the 160-node network, these were 320/sec, and 40,000.

We again show MMR and N-MMR as in the previous sub-section. Instead of showing these for a single service "s0", we show it averaged across all the 20 kinds of services "s0"-"s19". Fig. 5.12 shows the two ratios as a function of the number of overlay nodes. Fig. 5.13 shows the path length as a function of the number of overlay nodes.

We see that with a larger overlay size, the load variation shows an increase, but only a

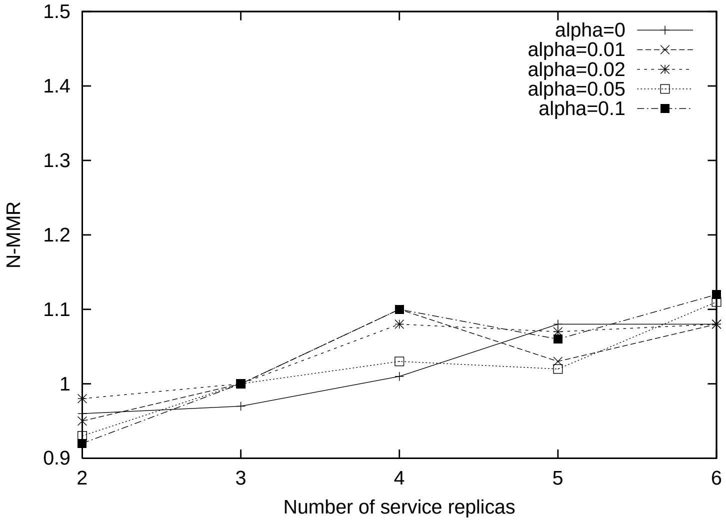

Figure 5.11: Next-to-max-min-ratio (N-MMR) for different values of α

small increase. The MMR measure has an average value of around 1.4, and the N-MMR measure metric has an average value of around 1.2, even in the case of 8 service replicas in the 160-node network. The path length remains more or less the same with increasing overlay size since we have the number of service replicas proportional to the overlay size.

5.3.4 Load balancing and failures

One of the primary goals of our architecture is the recovery of client path sessions on network failure. In Chapter 4, we studied the detection of failures, and recovery using alternate service replicas. We considered *end-to-end* recovery, where an altogether new path is established for each failed client session after an overlay link failure is detected. One of the concerns with path recovery is that a large number of client sessions may have to be restored when an overlay link fails. It is important that this process of restoration does not overload any particular service replica. Here we study the behavior of our mechanism when a large number of client sessions have to be restored.

We use an 80-node configuration for this experiment. The network has ten kinds of services ("s0"-"s9"), each with four replicas. The path creation rate is 80/sec, and the total number of paths created in the duration of the experiment is 20,000. As client path sessions are setup and torn down,

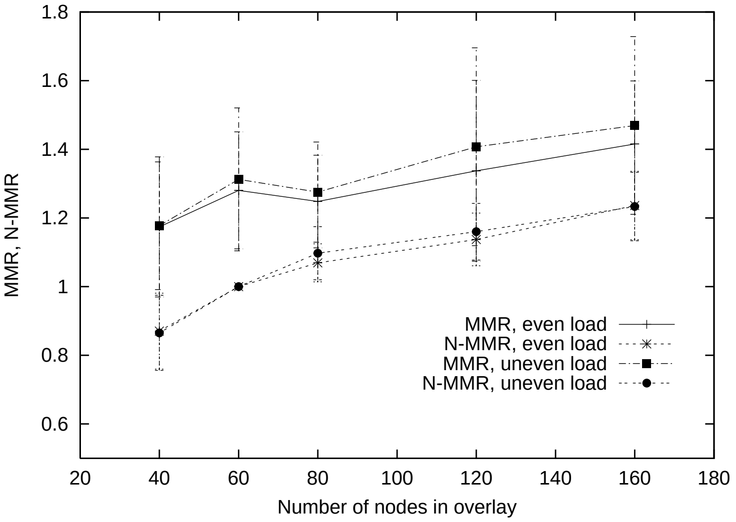

Figure 5.12: MMR, N-MMR for different overlay sizes

we introduce a deterministic failure in the overlay link that has the maximum number of client paths traversing it. The failed link is between nodes 12 and 20, and the failure happens around 243 seconds into the experiment. A total of 595 paths are recovered.

We show the load variation for two different services: one for which one of the replicas is present at a node that is at one end of the failed link, and the other for which none of the replicas are present at either end of the failed link. We choose services "s8" and "s0" respectively: the service "s8" has one of its replicas at node 20. Fig. 5.14 shows the load variation for the service "s8" and Fig. 5.15 shows the case of "s0". While we show the plots only for the services "s8" and "s0", the behavior for the other services are the same.

We make two observations: (1) In the case of "s8", as well as for "s0", the load for one of the replicas temporarily goes below the other three, and it catches up in a short period of time (20-30 sec), (2) The difference between the loads of the three replicas (that get used more), and the load of the single replica (that gets used less), is much more in the case of "s8" than for "s0".

The reason for the split in the load is the following. The entire set of paths that fail undergo recovery within about 1.5-2 seconds (see Chapter 4). This is simply the signaling time for the setup of the alternate paths. Our piggybacking mechanism's feedback loop is not fast enough to react

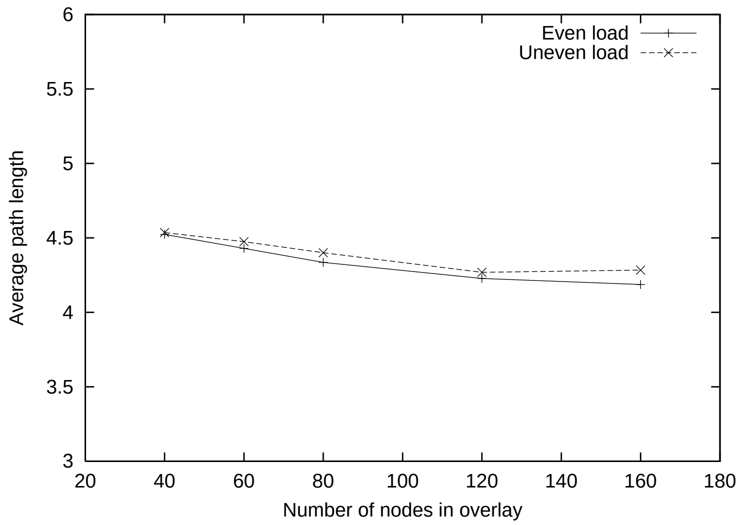

Figure 5.13: Path length for different overlay sizes

within this short period of time for the simple reason that the feedback loop itself takes the same time as the (alternate) path creation. The load of the replica that falls below the other three catches up over time since future client requests use this replica. The explanation for the larger difference in the case of "s8" is simply that in the case of node 20, a larger fraction of its service-level paths undergo failure recovery, since it is closer to the failure, than the case of the replicas of service "s0".

5.3.5 Simultaneous failures

We now show the effect of simultaneous failures on the load variation. The setup is similar to that in the previous sub-section except that we fail two of the most loaded overlay links this time: the one between nodes 56 and 58, and the one between nodes 29 and 35. There are a total of 1205 client paths that undergo recovery. The failures happen at around 247 seconds into the experiment. Fig. 5.16 and Fig. 5.17 show the load variation across the four replicas of "s5" and "s0" respectively, as a function of time. The service "s5" has a replica on node 35 (one of the ends of one of the failed links), while "s0" has no replica on any of the four nodes involved in the link failures.

The same two observations that we made in the previous subsection are valid here too: (1) one of the replicas is left behind during the load increase, and (2) the difference is larger in the

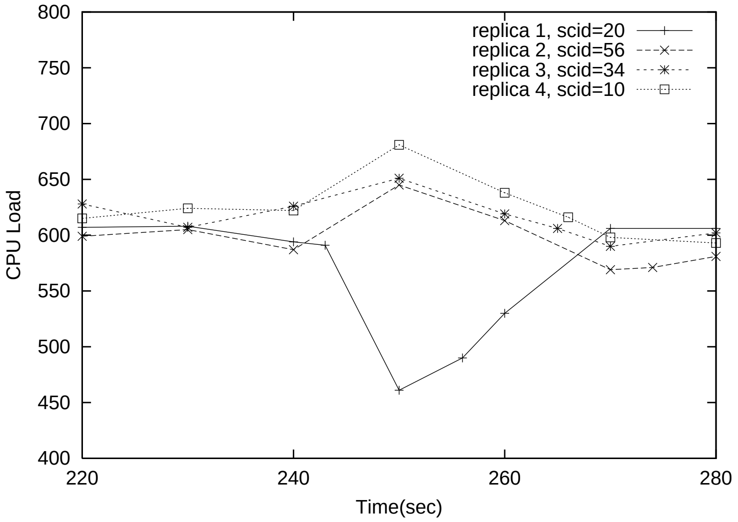

Figure 5.14: Load variation under failure/recovery: s8

case of the service with a replica close to the failure. The explanation for these remains the same too. One difference between Fig. 5.14 and Fig. 5.16 that can be observed is that the load at the replica that gets left behind remains flat for a longer time in Fig. 5.16 than in Fig. 5.14. A look at the plots for the other services (from the 20 services "s0"-"s19") for the experiment in the previous subsection as well as for the experiment here reveals that such variations in finer behavior do exist across the different services. Specifically, the difference between Fig. 5.14 and Fig. 5.16 is not due to the double-link failure in Fig. 5.16.

This difference is due to an implementation artifact – we tear-down a service-level path of a client session at its exit node immediately after switching the session to an alternate path (in case of failure). This causes the exit node to decrement its load immediately. But, the tear-down and the corresponding load decrement happen after a period of time (about 8 sec in our implementation) at the other upstream nodes. In the case of Fig. 5.14, node 20 happens to be an exit node for a larger fraction of the failed paths, whereas in Fig. 5.16, node 35 was an exit node for a smaller fraction of the failed paths. Hence the load for node 35 falls a little later. While we did observe such finer variations in the nature of the plots for the 20 different services, due to the dynamics of the system, the two observations that we made in the previous sub-section are valid across all the 20 services.

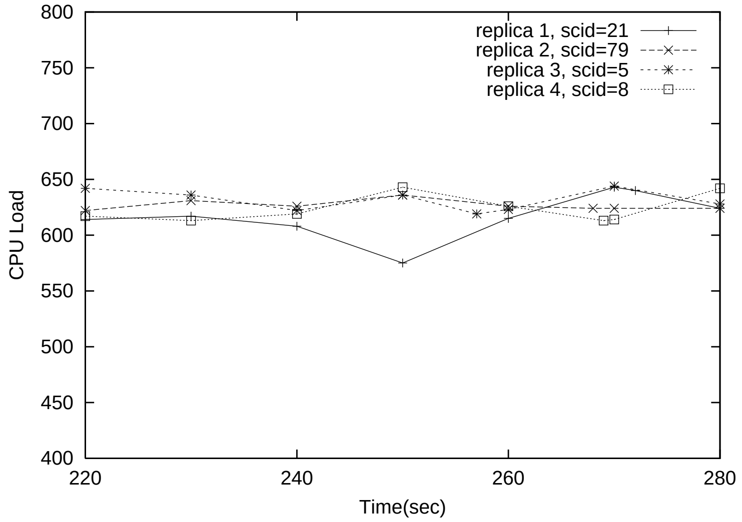

Figure 5.15: Load variation under failure/recovery: s0

5.4 Summary

One of the important goals of our architecture for wide-area service composition is that of performance of service-level paths. This means (a) appropriate choice of lightly loaded service instances and ensuring load balancing among replicas, and (b) choosing network paths with adequate performance. In this chapter, we have looked at the important issue of load balancing among service replicas in the context of composition. Service composition offers new challenges over traditional web-server selection since a *set* of instances have to be chosen for each client session, and since we are also concerned with failure detection and recovery during a client session. This leads to an altogether different architecture than the case of web-mirror replicas. We have an overlay network of service cluster execution platforms that participate in composition, load-balancing among themselves, and failure recovery.

We introduce the *least-inverse-available-capacity (LIAC)* metric for choosing service instances, as well as a piggybacking mechanism for quick feedback about server load. Piggybacking has several nice properties including low overhead, and an inherent mechanism to quickly correct load underestimates. We then introduce the no-op factor in the LIAC metric to avoid choosing far

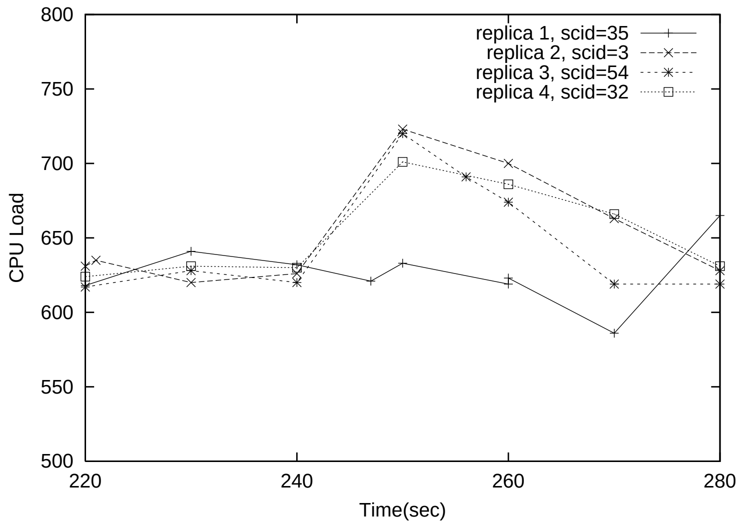

Figure 5.16: Load variation under simultaneous failure/recovery: s5

away service instances. We find through emulation experiments that the LIAC-NF metric combined with the piggybacking mechanism can perform well both in terms of load balancing and service-level path length in a variety of scenarios including single/double link failures.

In the previous chapter and this one, we have described our architecture and have presented various design studies to evaluate different aspects of it. In the next chapter, we turn to a description of a set of applications that use service composition in a heterogeneous network scenario. This illustrates the use of our architecture as well as presents an opportunity to evaluate its usefulness from the point of view of an implementation of a composed application.

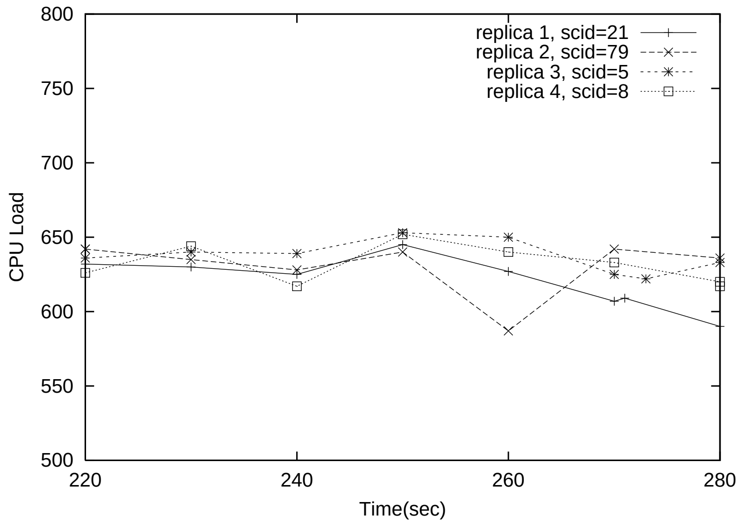

Figure 5.17: Load variation under simultaneous failure/recovery: s0

Chapter 6

The Universal Inbox: An Application of Wide-Area Service Composition

In this chapter, we present a set of scenarios that illustrate the use of our architecture for wide area service composition. We quantify how well the architecture works from the point of view of a composed application. The scenarios fall under the umbrella of the *Universal Inbox* set of applications. Universal Inbox is a metaphor for any-to-any communication in a device and network independent manner [67]. We developed and built the Universal Inbox as part of the ICEBERG project [81] that achieves service integration across multiple heterogeneous networks. The Universal Inbox achieves *personal mobility* and *service mobility* in an *extensible* manner. This is achieve through (a) generic data type transformation, (b) customizable redirection of incoming communication based on user preference profiles, and (c) device name mapping and translation.

A central piece of the architecture is the Automatic Path Creation (APC) service that achieves *data-type independence* through appropriate data transformations. The APC service uses *composition* of operators – service components that represent units of data transformation. The use of operators achieves extensibility in the APC service. The APC service can be directly instantiated on our architecture. This demonstrates the use of wide-area service composition.

We organize this chapter into the following sections. Section 6.1 presents the Universal Inbox architecture, describes how it achieves extensible personal mobility and service mobility. We focus on the APC service and also present the set of example scenarios of any-to-any communication that we implemented in our testbed. This brings out the use of the composition-based design of the APC service. Section 6.2 focuses on one particular application involving a composed text-to-speech application. We use this to study the application-level implications of false-positives in failure detection in our architecture. We then turn to an study of the usefulness of our architecture in terms of availability improvements. This study is from the point of view of the text-to-speech application, and uses a wide-area testbed.

6.1 Universal Inbox: Extensible Personal Mobility and Service Mobility

The concept of Personal Communication Services (PCS) comes from the telecommunications domain [64]. It consists of data and communication services with three kinds of mobility: terminal mobility, personal mobility, and service mobility. Terminal mobility is the ability of the user to communicate independent of his/her physical location. Personal mobility is the ability to redirect communication across heterogeneous user devices. Service mobility provides access to services independent of the user's end-point; i.e., the user sees the same set of services from all end-points.

With new communication devices and services emerging at a rapid pace, today's user has a range of communication end-points. A user may have several devices (cell-phone, pager, PSTN phone, desktop at office, etc.) and services (e-mail, voice-mail, instant messaging, information access services). Personal mobility and service mobility are important features in this context. The user wishes to receive incoming communication independent of the device in use (personal mobility), and access services independent of the device and access network (service mobility). Further, it is important to enable these features in an extensible fashion. That is, it should be easy to add emerging communication services, and integrate them with *all* of the existing ones. The right component functionalities should be provided so that this process of integration involves minimal development, and minimal deployment.

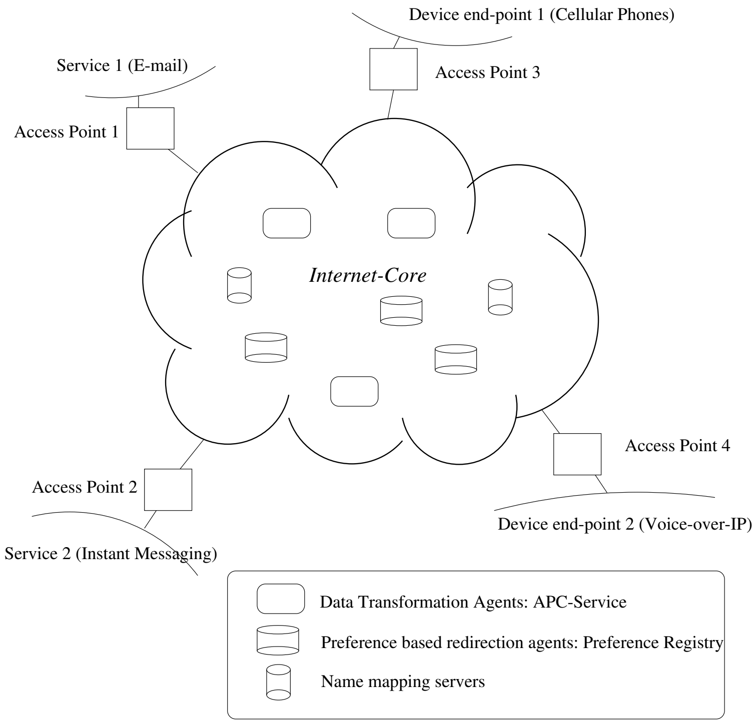

Figure 6.1: Universal Inbox: functional components in the Internet

The Universal Inbox achieves such extensible personal mobility and service mobility through three functional capabilities: (a) any-to-any data transformation – for accommodating diverse devices with different data formats, (b) storage and processing of user preferences – for ubiquitous redirection of incoming communication, and (c) device name translation and mapping – to handle the heterogeneous name spaces like cell-phone numbers, pager numbers, IP-addresses, etc. These capabilities are implemented as infrastructure services on the Internet. Thus the various networks are integrated with an Internet-based architecture. This is illustrated in Figure 6.1. We describe each of the functional components briefly now.

6.1.1 Data Transformation Service: Automatic Path Creation

In personal mobility or service mobility scenarios involving heterogeneous devices or services, some form of data transformation is required. We separate this functionality into an independent component that can be reused by all mobility features. This component is the *data transformation service*. This service achieves *data-type independence* in personal mobility and service mobility scenarios.

The data transformation component leverages composition. An *operator* is a generic data transformer (e.g., an audio codec), and a *path* is a series of operators strung together [49]. This directly maps onto our definition of a service-level path, with the operators being the independent service components. (In fact, the notion of service-level path in our work emerged from the design of the Universal Inbox). The data transformation service is thus called the *Automatic Path Creation (APC)* service. An example path is depicted in Figure 6.2. The path shown consists of two operators (the bold lines) – it converts a given piece of text to PCM audio.

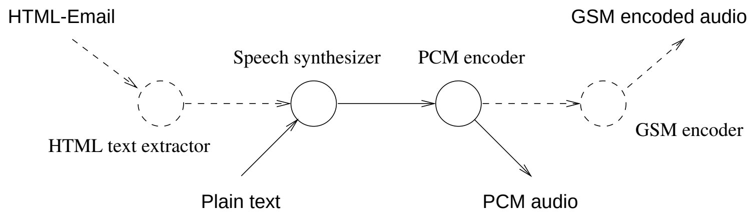

Figure 6.2: Illustration of an APC Path

The term "automatic" refers to the fact that we only need to specify the input and output data formats (and input source and output sink) to the APC-Service, it then strings together the operators necessary for the transformation "automatically", through composition of the appropriate operators.

This provides a highly extensible model. For instance, in Figure 6.2, suppose one wishes to add support for GSM audio as well, all that needs to be done is to add an operator (see the dotted lines on the right). Similarly, suppose we want to add support for conversion of HTML e-mail messages, we would just have to add the appropriate operator (see the dotted lines on the left).

A generic data transformation component is absent in today's communication infrastructure. Transformations, if any, happen at the edges (e.g., at the Inter-Working Function between the GSM and PSTN networks [60], or at each service provider deploying integration services) and are not generic enough or reusable for other services.

6.1.2 Preference Registry for redirection

The second functionality common to personal mobility scenarios is redirection. Redirection is often highly user-specific. This functionality is implemented within the Preference Registry (PR) component. The PR stores and processes user preference profiles and acts as the redirection agent across heterogeneous devices. While in today's communication infrastructure, the caller decides where and how to reach the callee, the preference registry achieves *control to the callee*.

The user's preference specifies the way a particular incoming communication should be handled. It is a function of several factors like the time of the day, caller-id, user location, user state, and so on. We represent user preference as a set of rules. We model it as a script that is processed by the PR, each time with a new set of input values to the script. Figure 6.3 shows an example of a user preference script.

IF (9AM < hour < 5PM) THEN Preferred-End-Point = Office-Phone; // At Office

IF (5PM < hour < 11PM) THEN Preferred-End-Point = Home-Phone; // At Home

IF (11PM < hour < 9AM) THEN Preferred-End-Point = Voice-Mail; // Sleeping

Figure 6.3: A Simple Preference Script

6.1.3 Naming Service

The Naming service is a component required for handling heterogeneous devices and services that have different name spaces. It allows us to map between the different user end-point identities in these name spaces. We define a unique-id for a user which is equivalent to MPA's

PoID [71] or UMTS's UPT number [64].

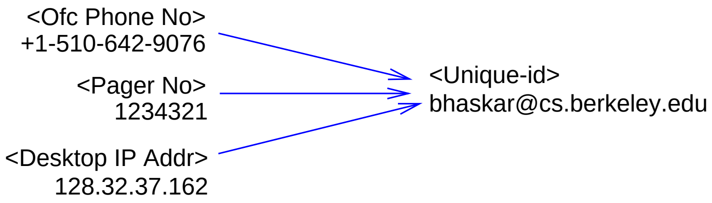

Figure 6.4: Examples of Name mappings

The Naming service maps between the identities of the user's different end-devices, or communication services. All these identities map to the user's unique-id. An example is shown in Figure 6.4. This mapping allows lookup of further information indexed on the user's unique-id. For instance, the location of the user's preference registry is determined as a mapping from the unique-id. The Naming service is used as the bootstrap mechanism for locating a user or a service end-point. Hence it needs to be globally distributed and scalable. We use a DNS-like tree structure for the name tree [67]. We exclude the details here since they are not relevant to our discussion.

6.1.4 Access Points

An *Access Point (AP)* provides gateway functionality for an access network. It exports a generic session setup interface to the Internet core. The common session setup interface provides a level of indirection that is key to achieving network independence. An AP could interface to a service or to devices in an access network. For instance, we could have an AP to interface with an e-mail store, and another at the Mobile Switching Center (MSC) [66] of a GSM network to interface with cellular-phones.

6.1.5 Putting it all together: an example

Figure 6.5 shows a simple personal mobility scenario using the architectural components introduced above. It shows how personalized redirection would work across heterogeneous device-types. The caller dials a number from the cellular network. The AP at the edge (e.g., at the MSC or the Base-Station) intercepts this (step 1) and gets the unique-id and the location of the preference registry of the callee using the distributed Naming service (step 2). It then gets the current preferred end-point of the callee from his PR (step 3) and establishes a call session through another AP to the callee (steps 4-7). The APC service is used for any data transformation (step 8).

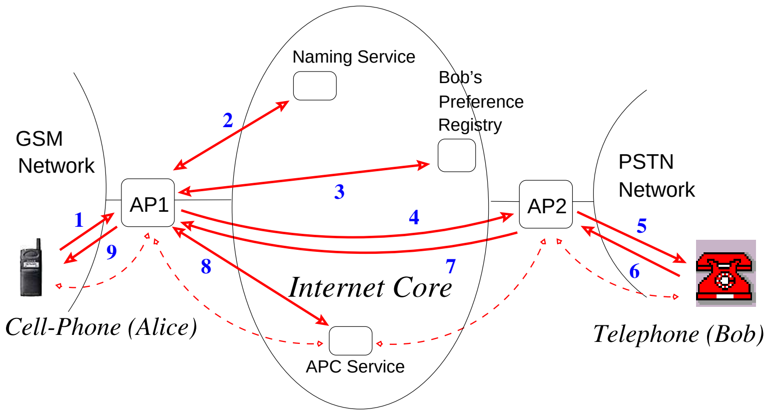

Figure 6.5: Putting it all together: An Example

6.1.6 The Universal Inbox and Service Composition

The Universal Inbox uses the APC service which in turn uses composition. The composition involves operators, which are data transformation service components. This can directly be instantiated on our architecture for service composition. The availability and performance features achieved in our architecture are directly inherited by the Universal Inbox communication scenarios. The other components of the Universal Inbox provide the necessary functionalities to implement the control mechanisms for the various communication services. We next present a wide variety of communication scenarios in the Universal Inbox that we have implemented. This brings out the extensibility features, a crucial part of which is the flexible APC service.

6.1.7 Implementation experience: Universal Inbox scenarios

Figure 6.6 and Table 6.1 show the step-wise addition of end-points and features to the Universal-Inbox. Specifically, they show the Access-Points, and operators at the APC service that

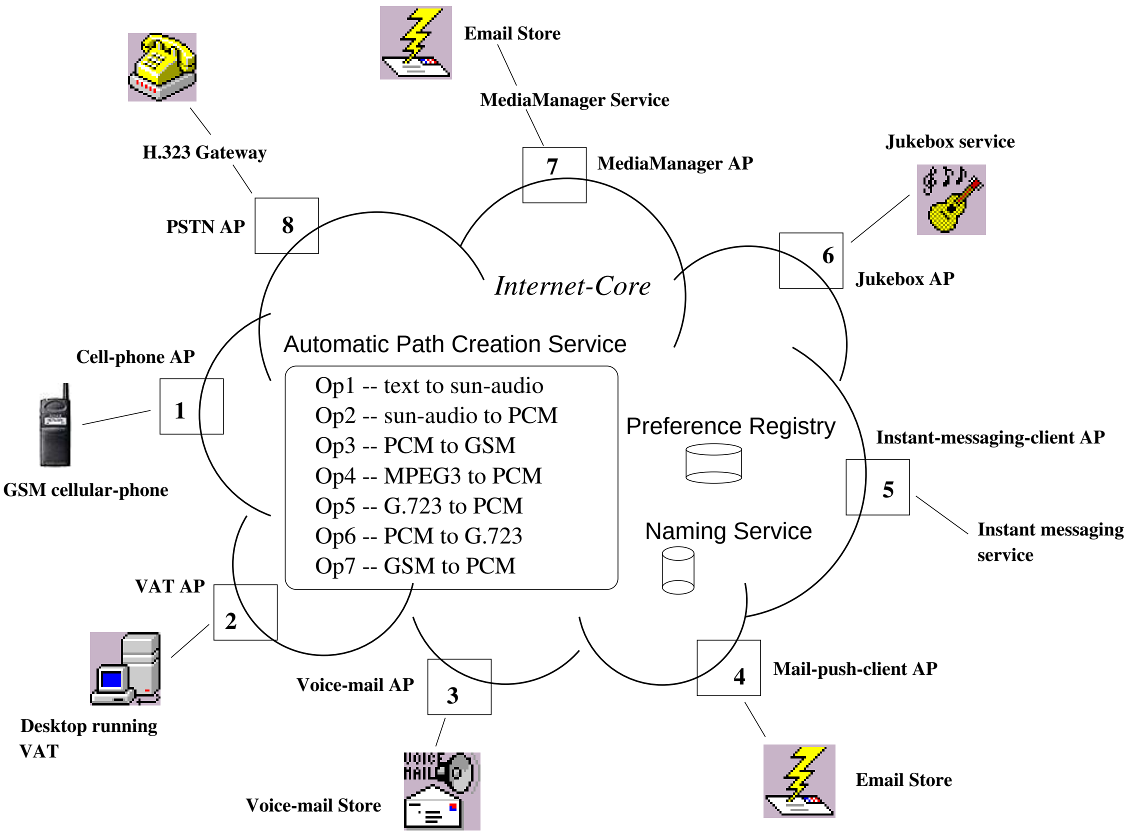

Figure 6.6: Step-wise additions to the Universal-Inbox (refer Table 6.1)

		Device/Service AP	Operators in APC	Personal/Service mobility features
1		Cell-phones (#1) & Voice-over-IP (#2)	(none)	Call redirection/screening based on time-of-day & caller-id
2	(+)	Voice-mail (#3)	(none)	Call redirection to voice-mail also possible. Voice-mail access from cell-phone/VoIP end-points
3	(+)	Mail-push-client (#4)	Op1 (text to sun-audio) Op2 (sun-audio to PCM) Op3 (PCM to GSM)	Email redirection to cell-phone/VoIP/Voice-mail
4	(+)	Instant-message client (#5)	(none)	Instant message redirection to cell-phone/VoIP/Voice-mail
5	(+)	Jukebox-service (#6)	Op4 (MPEG3 to PCM)	Jukebox access from cell-phone/VoIP
6	(+)	MediaManager service (#7)	(none)	MediaManager access from cell-phone/VoIP
7	(+)	PSTN end-points (#8)	Op5 (G.723 to PCM) Op6 (PCM to G.723) Op7 (GSM to PCM)	Call redirection to PSTN, E-mail redirection to PSTN, Instant-message redirection to PSTN, Jukebox access from PSTN, MediaManager access from PSTN

Table 6.1: Step-wise additions to the Universal-Inbox (refer Figure 6.6)

were added. During each step, additional preference profiles and naming entries were also created. The "+" at the beginning of each entry in Table 6.1 denotes that the entry represents an *addition* to the set of end-points, operators, and features. All of the APs in the table have been implemented.

To start with (row #1 of Table 6.1), we have two kinds of end-points in implementation: (a) GSM cell-phones, and (b) Desktop voice-over-IP end-points in the form of the Visual Audio Tool (VAT) [47]. Each kind of end-point is reachable through an Access Point. The GSM AP interfaced to the cell-phones through a Base-Station and a BSC/MSC simulator; the details of the interface are not relevant here. The VAT end-points use the GSM codec and we have a single data type in our system: GSM audio.

The name hierarchy has entries for IP-addresses and cell-phone numbers. Personalized redirection across the heterogeneous end-points is possible through the use of the PR and Naming service components. The example in Section 6.1.5 showed how this would work.

Extension of the redirection functionality to include voice-mail involves the addition of an appropriate AP (row #2). This AP also allows a user to access her voice-mail through either of the earlier two end-points. Note that the development and deployment of this voice-mail AP is independent of the previously existing APs.

The first text-based end-point we integrate is e-mail (row #3). For this, we implement a client Access Point that resides at the user's e-mail store and establishes outgoing sessions for reading out e-mail to the user's preferred end-point. For each e-mail received in the user's inbox, the Access Point checks the user's Preference Registry to see if (a copy of) the e-mail should be redirected to another end-point (e.g., the user's cell-phone).

The support required for this at the APC Service is three operators for text to speech conversion: (a) text to sun-audio (based on festival [13]), (b) sun-audio to PCM (based on the sox Unix program), and (c) PCM to GSM (based on the toast GSM codec [39]). With this in place, e-mails can now be redirected to *all* previous end-points.

Next, we do a similar integration with an instant messaging service, by implementing an AP for the same (row #4). The AP interfaced with the Sanctio instant messaging service that was developed as part of the Ninja project [15]. We can now reuse all of the APC functionality added in the previous step. The functionalities available with e-mail earlier are now readily available with instant messaging as well.

We now enable access to two services in turn. The first is the Jukebox service (row #5) – which was also developed independently as part of the Ninja project [15]; the service plays streaming MPEG3 encoded music to the user's desktop. We add an AP to proxy for this service, and also add an operator at the APC service (MPEG3 to PCM converter, based on the mpg123 unix program). We reuse the PCM to GSM operator added earlier, and access to this service is now enabled from the device end-points: cell-phones and VoIP desktops.

The second service we enable access to is the MediaManager service (row #6). This was also developed independently as a separate project. It is a service that sits in front of the user's e-mail store, and is capable of doing intelligent processing (such as summarizing) on the e-mail. Enabling access to this involved building the AP to proxy for it. The MediaManager is capable of outputting several audio formats including the GSM format. Hence no additional operators are required at the APC service.

Finally, we consider how we can add PSTN telephone end-points to the system (row #7). We have a H.323 gateway [1] interfacing with the PSTN network. We add an AP in front of this gateway. This gateway supports only the G.723 audio format. To inter-operate with GSM-based end-points, we need to add the three operators shown in the table. With this done, all of the previous functionalities: redirection, screening, and service access that were possible with the earlier end-points, will now be possible with PSTN telephones as well.

Discussion of Extensibility Features

In general, extending the system to new devices or services involves the addition of an Access Point. An Access Point essentially provides the glue for integration. It has a device- or service-specific part, and it has a generic part. The generic part is the Internet session establishment protocol, such as SIP [75]. In implementation, we have used a simple Java RMI based session initiation and termination protocol, since this is not the focus of our work.

The device- or service- specific part of the AP could be quite complicated. For instance, developing this for the GSM AP required understanding the GSM protocol stack and was a considerable effort of about nine person-months. However, APs to simple services can be developed easily. The interfaces to the Jukebox service and the MediaManager service are quite simple – JavaRMI calls to retrieve songs and messages respectively. APs to these services are only about 700 lines of Java code each – one could consider adding more frills to these APs though.

While extending the system, in some cases, when there are new data formats involved, operators have to be added to an APC service instance. Here again, an operator may not be easy to implement by any means. For example, a text to speech conversion software is non-trivial. But the key is that such functionalities can be reused – not only in implementation, but also in *deployment*. That is, a third party APC service in the Internet-core can be used for the appropriate conversions. This is richer than the reuse of functionality that is possible with other architectures like the MPA [71]. An MPA Personal-Proxy (PP) can reuse the “conversion drivers” for different mobility features. But such reuse is not possible across two instances of the PP.

The deployment of new services and integration of new devices is helped by the fact that the access points are *independent*. They can be developed and deployed at different points in the infrastructure, by different parties.

6.2 The Text-to-Speech Composed Service

In this section, we describe how the text-to-speech composed service would be instantiated on our architecture. Section 6.2.1 describes the relevant details of the implementation of this composed service. We use the implementation in our testbed to evaluate our architecture from the point of view of this application. First, in Section 6.2.2, we study the effect of recovery and false-positives in failure detection on the application. Next, in Section 6.2.3, we use a wide-area testbed to evaluate the usefulness of our recovery algorithms from the application perspective. We measure the improvements in availability due to the quick recovery mechanisms using measurements over the Internet.

6.2.1 The Text-to-Speech composed service on our architecture

We have implemented a composed text-to-speech service in the framework of our architecture. In this scenario, two services are composed: a text-source, and a text-to-audio service. Figure 6.7 illustrates this. The text-source could be a user's email service (like mail.yahoo.com), or a news source, etc. In implementation, we have a simple text-source that serves sentences from a file. The text-to-audio service is implemented using the Festival Speech Synthesis System from University of Edinburgh. The audio format we choose for output is PCM U-Law 64kbps – the same as that used by the PSTN. We have also implemented the "no-op" service that serves as a data-switch, without transforming the data. Figure 6.7 also shows the different data/control exchanges (D1, D2, D3, D4), which we explain now.

The data and control exchanges are done after path establishment. Path establishment itself in our architecture involves the steps of (a) client making a request for the required composition to the cluster manager of its exit-node, (b) the exit overlay node choosing the service instances and

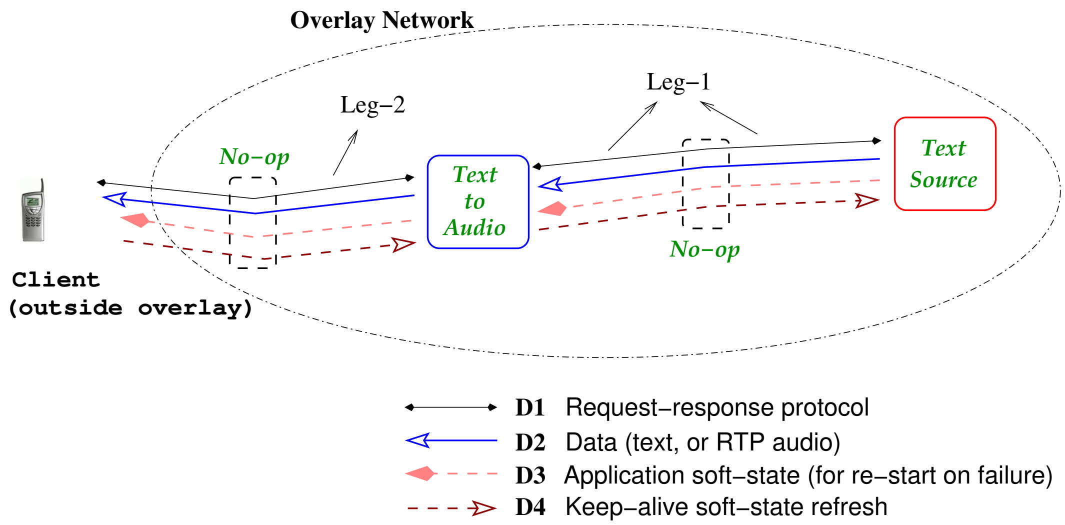

Figure 6.7: The text-to-speech composed service

the service-level path to use based on the availability and load balancing constraints as explained in the previous chapters, and (c) the exit overlay node sending signaling messages upstream along the path to instantiate the service instances and the no-op services.

After path establishment, the application-specific communication begins. In this example, the end-client requests for a specific piece of text, using an application specific protocol, which in this case is a simple request-response protocol (D1). Since, in our model, the user would have subscribed to the portal provider composing the services, the security keys setup during subscription can be used for authentication, as well as for data encryption.

Once the request is made, in our particular implementation of the service, the text-source supplies the text-to-audio service with successive sentences (D2). The audio is then streamed by the text-to-audio service using RTP (D2) – this audio can be heard on desktop speakers at the end-client in our implementation. Although we have not integrated this service with any real phone system, this would be an easy step, given the extensible personal mobility and service mobility mechanisms in the Universal Inbox, as described in Section 6.1.

State Management: The application itself could have soft-state (as stated earlier, we assume that the application does not have "persistent" state that should last across sessions). In our application, this soft-state consists of (a) the position in the text source currently being processed by the text-to-audio piece, (b) the RTP stream index currently being received by the end-client, and (c) any intermediate buffers at the text-to-audio service while it is in the middle of processing a sentence.

The soft-state in (a) and (b) is essential for determining the position from where to re-start, in case the service-level path is re-instantiated on failure. And the state in (c) – we simply discard this and re-start processing at the current sentence at the new instance of the text-to-audio service. The soft-state in (a) and (b) is periodically (every 500 ms) sent to the end-client, along with the data stream, *downstream* (D3). This state is then used to re-start the service-level path session when it is re-instantiated.

At the application level, we also have a periodic soft-state refresh that goes *upstream*, from the end-client to the data-source (D4). Just like in the case of the "connection-state" at the cluster-managers, this soft-state refresh means that we need not ensure that the tear-down messages traverse through the old path, in case we switch to a new one.

6.2.2 Text-to-speech composed service: evaluating failure recovery

In this section, we present an evaluation of the effects of the path recovery mechanism from the point of view of the application we have implemented.

Experimental Setup

We use the emulation platform for this initial evaluation. In this setup, we us a 20-node overlay network, generated as described in Section 4.3. We place a single instance of the text-source service as well as the text-to-audio service in the overlay network. The overlay node for the placement of each of these services is chosen at random. We establish a client session of the composed text-to-speech service in this setup. (We tried the experiment with different paths in the overlay graph, and the results were similar). An overlay link along the service-level path is then deterministically failed in the middle of the client session, in a controlled fashion, by appropriately configuring the emulator software. We then measure the gaps between the receipt of successive audio packets at the end-client. This metric captures the interruption the user would experience when a network failure occurs.

Results

Table 6.2 shows the “gaps” seen at the end-client due to the failure, for different scenarios. The recovery time of 2,963 ms, after leg-2 is failed, has three components: (a) about 1.8 sec of failure detection time (Chapter 3), (b) about 700 ms for the setup of the alternate service-level path, and (c) about 450 ms for re-instating the application state. Since component (b) is relatively small, the propagation delay in the network is not actually a major factor in determining the time-to-recovery. The third component is specific to the application, and in this case is the time required for re-processing the current sentence being read out to the user. This is the time taken to re-instantiate the application-level soft-state in the new service-level path.

Scenario	Gap seen at the end-client
Failure of leg-2, with recovery	2,963 ms
Failure of leg-2, no recovery	10,000 ms
Failure of leg-1, with recovery	822 ms

Table 6.2: Gaps seen at the end-client

The table also shows what the effect would have been, had there been no recovery algorithm – the client would have seen a gap as long as the failure duration – 10 sec in this case. In the case of failure of leg1 (3rd row), even though the detection itself takes 1.8 sec, even as the alternate path is being setup, the original path continues to stream data, from the text-to-audio instance, downstream. This is due to the one-sentence buffer at the text-to-audio service instance. Thus, recovery time as perceived by the end client is much lower.

Discussion

The application we have studied is a real-time application, but not an interactive one. Our recovery schemes work extremely well in such cases – delays of 3 sec can be completely masked from the user with the use of buffering. In fact, with our recovery mechanisms, the amount of buffering required might be lesser than otherwise. Such quick recovery is possible since we do not rely on the failure information to propagate and stabilize across the network, to effect recovery.

For interactive applications, such as two-way audio, our recovery algorithm offers better behavior when it detects failure correctly. In fact, if it avoids long outages lasting several tens of seconds, this greatly improves availability. However, in the case of false-positives, use of path recovery can cause slightly longer interruptions than otherwise.

The text-to-speech application represents a case where there is a significant amount of application level soft-state, in terms of the processed text, in audio form, of the current sentence being streamed. Audio transcoders are likely to keep much less soft-state in terms of buffers.

Effects of false-positives: In our failure detection mechanism, there could be false-positives: when we timeout and conclude a failure, but the failure was intermittent/short. This has two effects. Firstly, additional overhead at the new service instance, to rebuild its soft-state – in our example application this is the processing that has to be done to the current text-sentence being streamed, as well as the buffered sentences. Since our trace-study mentioned in Chapter 3 have shown that false-positives occur very infrequently, once an hour at maximum, this is minimal additional overhead. Secondly, there is possible additional delay at the end-client, during path switch-over. This delay is of the order of 1.1 sec or less (2,963 ms, less the actual outage time of at least the timeout period, 1.8 sec). For an application like ours, this can be easily masked by buffering at the end-client.

Scaling: So far we have discussed the performance of a single service-level path. When scaling the system, the control overhead of our middleware software should also scale. One of the possible bottlenecks that we examined in detail in Section 4.3 was the message processing overhead when a failure is detected between peers, and a whole set of paths running between those have to be recovered. Our evaluations showed that a single cluster-manager can support of the order of 250 simultaneous client sessions, while still managing the message processing overhead when a set of client path sessions have to be recovered simultaneously.

In our implementation of the text-to-audio service, a single node, of similar processing power as for the cluster-manager, can support about 15 simultaneous clients. We estimated this by examining the amount of processing required for processing a sentence, and the amount of PCM-audio it represented. Hence an additional cluster-manager is required for every $350/15 = 23$ nodes implementing actual text-to-speech services. While this is a crude estimate, it points to the fact that the additional provisioning overhead required for the middleware functionality of our architecture is minimal.

6.2.3 Wide-area experiments: evaluation of availability improvements

In this section, we turn to an evaluation of the usefulness of our recovery algorithms. We use the composed text-to-speech application in a wide-area testbed, and estimate improvements in availability using long-running experiments.

Wide-area testbed setup

We run these set of experiments in a real wide-area testbed. Thus all network conditions are real, and not emulated. We configure an eight-node overlay network as shown in Figure 6.8. There are four nodes within the continental US. These are university sites, and are inter-connected via the Internet2 backbone [6]. There are two nodes behind commercial ISPs. We have a node connected via a cable modem at Berkeley, and a node connected via DSL at San Francisco. In addition to these, we also have two nodes at two other continents: one at UNSW, Australia, and another at TU-Berlin, Germany. The overlay network configuration is as shown in the figure.

Each node acts as an overlay node service cluster, although we only have one physical machine at each location. Each node thus runs the cluster-manager software. In addition, the text-source service, and the text-to-speech conversion component have two, and three instances at the locations shown in the figure. Each node is also capable of instantiating a no-op service instance.

We have clients attached to the overlay node at Berkeley, as well as the overlay node at CMU. That is, these clients have their exit-nodes as the overlay node at Berkeley, or CMU respectively. We have equal number of clients at either of these overlay nodes. In each set of clients, we designate half the set to have recovery algorithm enabled, and a other half to have recovery algorithm disabled. That is, half the client sessions enjoy the benefits of recovery, and the other half

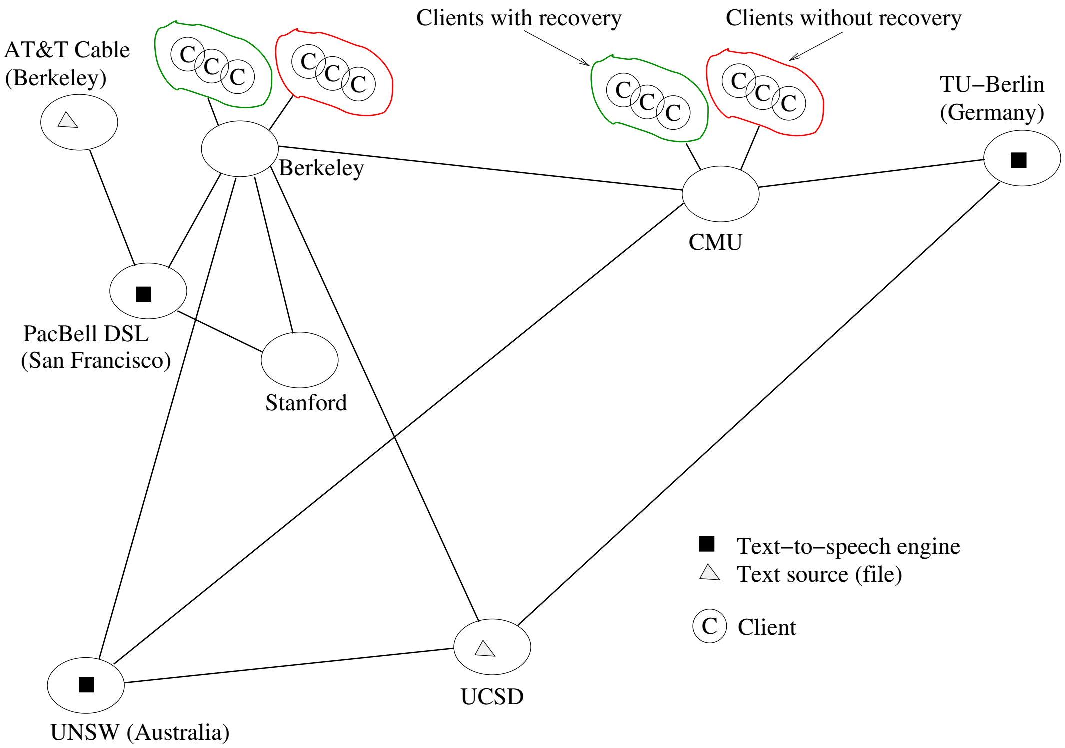

Figure 6.8: The wide-area testbed

function without the recovery algorithm, thus relying upon Internet path recovery.

The Internet path latency on the different overlay links varied all the way from about 3 ms (between Berkeley and Stanford), to about 220 ms (between Berkeley and UNSW). The number of Autonomous Systems in the Internet path representing an overlay link varied between two (Berkeley to Stanford) and six (Berkeley and UNSW).

Experiments and Results

We have four paths in the system at any point of time. Each path with recovery enabled has a counterpart with recovery disabled, running at the same time. We setup both the client sessions to take the same service-level path, although our load-balancing mechanisms may dictate otherwise. This allows us to compare the two and estimate the improvement in availability due to the recovery algorithm. Of the four paths, two are for clients at Berkeley, and two for clients at CMU. The duration of each composed text-to-speech session is 150 sec (2 min and 30 sec). A new client session is started up for each session that ends, thus maintaining four paths in the system at all times. The experiment runs for one day at a time, for 11 days. This includes weekdays as well as weekends.

We choose an application-level metric to characterize the behavior of the system. We compute the loss-rate as seen at the end client, measured in five-second intervals. We capture this metric at three different levels, and present results accordingly:

1. From the point of view of a single pair of service-level path experiencing failure (one service-level path with recovery enabled, and its counterpart with recovery disabled).

2. Considering all failed service-level paths.

3. Overall improvement in availability taking into account *all* service-level paths – those that failed and those that did not as well.

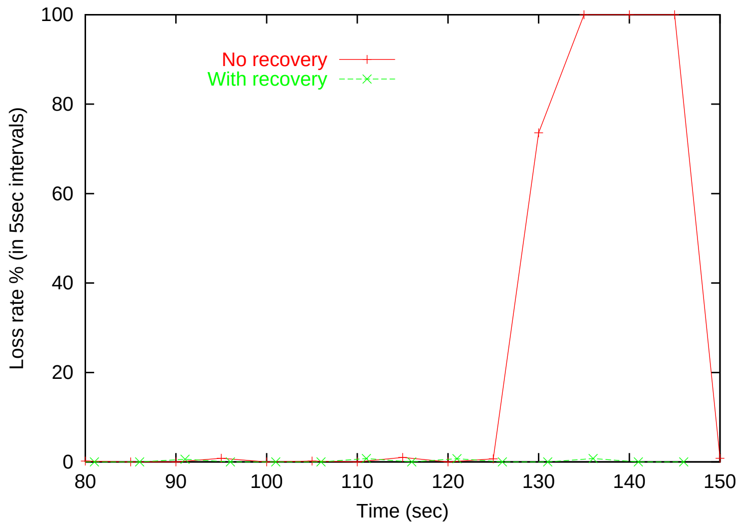

Figure 6.9: Effect of recovery algorithm: result for a pair of paths

Figure 6.9 shows the results form the point of view of a pair of paths. The x-axis shows the time since the beginning of the session (for either path), and the y-axis shows our metric, the loss-rate measured in five-second intervals. Both the service-level paths start at the same time, and have the same service-level path. At around 125 seconds into the session, there is an Internet path outage. The path with recovery enabled is able to detect the failure and recover from it choosing an alternate service-level path. The client session with recovery disabled relies on Internet path recovery and experiences a long outage lasting about 20 seconds in this case.

In this case, the failure was in leg-1 of either service-level path. The one with recovery enabled was able to recover without experiencing any outage at all, as explained in the previous section (see Table 6.2). We also observed several cases where the service-level paths experienced failure in leg-2. We now present the results from the point of view of all paths that experienced an outage.

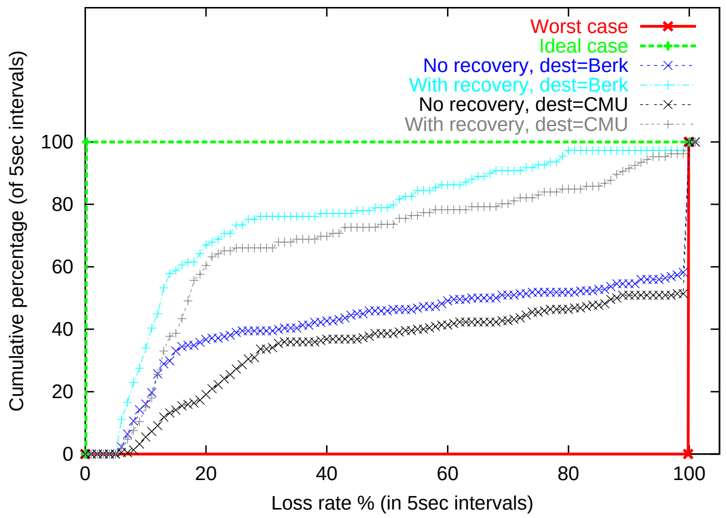

Figure 6.10: Effect of recovery algorithm: result for all service-level paths that experienced an outage

Figure 6.10 shows the performance as seen by all service-level paths that underwent a failure. We consider all the sessions that were setup in all the eleven days of experiments for this plot. The metric is the same: loss-rate as seen in five-second intervals. We consider all five-second intervals in all the service-level paths that experienced failure, and compute the loss-rate as seen in all these intervals. We plot a CDF of these loss-rates. The figure shows such a CDF for six cases: an ideal case, a worst case, clients at Berkeley with recovery, clients at Berkeley without recovery, clients at CMU with recovery, and clients at CMU without recovery.

In the ideal case, there is 100% of the five-second intervals have 0% loss rate. In the worst-case, 100% of the five-second intervals have 100% loss rate. The experimental values fall between these two extremes. The two extremes are shown to point out that the closer one gets to the ideal case, and further from the worst case, the better.

Although we have plotted the data for Berkeley and CMU in the same graph, we do not intend to compare these. The comparison is between the clients that have recovery enabled, and those that have recovery disabled. We see that both in the case of clients at Berkeley as well as clients at CMU, the clients with recovery algorithm enabled are able to perform much closer to the ideal case. For instance, for the clients at CMU with recovery disabled, over 50% of the five-second intervals experienced over 90% loss-rate. However, for the clients with recovery enabled, only about 10% of the five-second intervals had over 90% loss-rate.

Note that the above data is shown only for the set of service-level paths that actually experienced failure. For a calculation of the overall improvement in availability, we need to consider *all* service-level paths – those that experienced failures, and those that did not as well. A large percentage of service-level paths actually do not experience any failure at all, and the loss of availability is due to the fraction that does experience failures. We now present the data for the overall improvement in availability.

We first define an application-level notion of availability. For each five-second interval in a client session, we define the system to have been available for that client during that interval, if the loss-rate seen in that interval was less than 20%. Otherwise, that five-second interval is counted as having been unavailable. Looking at all five-second intervals in all the service-level paths setup in a day, we compute the overall availability of the system. We compute this for the client sessions with recovery and those without recovery, and make a comparison.

Figure 6.11 shows a bar plot for the clients at Berkeley. It makes a side-by-side comparison of the percentage *un*-availability (100% - percentage availability) for the client sessions with and without the recovery algorithm. The bar plot shows the data for each of the eleven days.

We first note that the unavailability percentage is quite low for all the cases, in absolute terms. That is, availability is quite high, over 99.5%. However, this is nowhere near the desirable

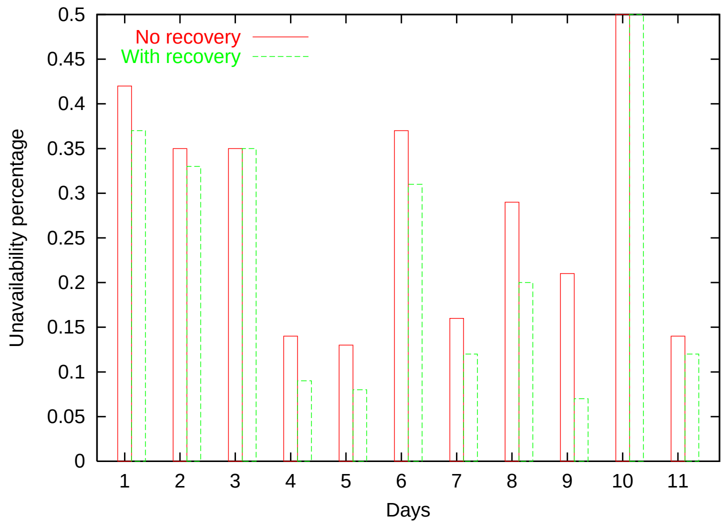

Figure 6.11: Effect of recovery algorithm: availability improvement for clients at Berkeley

availability standards. Recall that Telephone networks achieve 99.999% availability.

There are several days on which there is little or no improvement in availability. This is due to the following reason. Our testbed overlay network is small and does not have rich connectivity. For instance, Berkeley and CMU have only one connection to the rest of the Internet. Thus for network failures close to these nodes, there may not be an alternate service replica or an alternate service-level path that is reachable. Hence our recovery algorithm is not able to work around such failures.

However, there are several cases where our recovery algorithm is able to improve the availability by huge amounts. On days 4, 5, and 9, the unavailability fell by about a factor of two. This is especially significant since the percentage unavailability is already quite low. Our recovery algorithm is thus able to push availability much closer to 100%.

Figure 6.12 shows the bar plot with unavailability percentages for five days for the client sessions at CMU. (The unavailability for the other days was over 1.5%. For clarity, we do not show these in the plot). On days two and three, the improvements in availability are especially dramatic. On day two, the unavailability fell from 0.27% to 0.04%, a factor of about seven. And on day three, the unavailability fell from 0.21% to 0.02%, a factor of over ten, with the use of our recovery

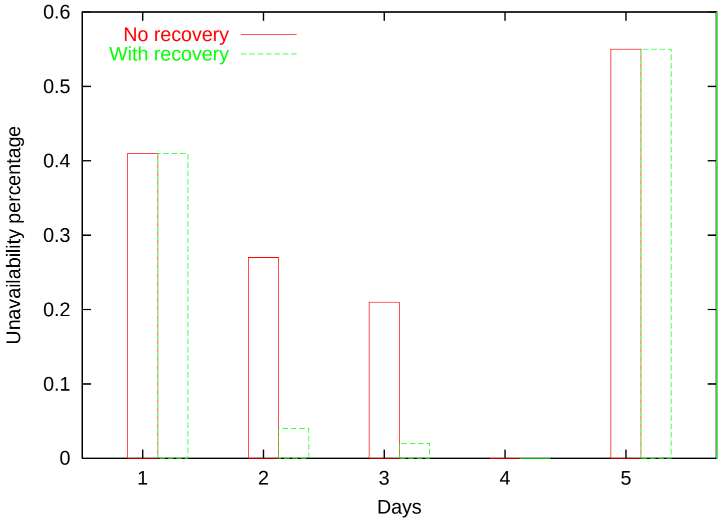

Figure 6.12: Effect of recovery algorithm: availability improvement for clients at CMU

algorithm. Again, this is especially significant since the availability is very good to begin with, even without our recovery algorithm. We are able to push the availability much closer to 100% with our failure detection and recovery mechanisms.

6.2.4 Summary and discussion

In the previous two subsections, we have presented an evaluation of our architecture from the point of view of a composed application that we implemented as part of the Universal Inbox architecture. An emulation-based evaluation allows us to look at the effect of the presence of application-level soft-state during failure recovery. Restoration of this soft-state incurs a small cost in terms of the addition to the end-to-end time-to-recovery. This is less than 500 ms for the text-to-speech service component. A false-positive from the point of view of the application has little effect on the performance, except for the additional work done in building the soft-state at the new service instance. Since false-positives happen rarely in absolute terms, this is not a huge overhead.

We evaluated the architecture for its effectiveness in terms of improvements in availability. We used a wide-area testbed for this purpose, and used the text-to-speech composed application. We found that the architecture is able to achieve great improvements in availability through quick failure detection and recovery. The unavailability comes down by factors of up to 2-10 in many cases.

In some cases, we are not able to improve the availability since the failure happens in such a way that there is no alternate service replica that is reachable. However, we believe that this is a limitation of our testbed, and with a better placement of the service cluster nodes on the Internet, and with better connectivity in the overlay network, this factor would reduce. This points at an important and interesting avenue for further exploration: the placement of the service-cluster overlay nodes for best connectivity.

We finally note that in our estimates of improvements in availability, we have considered the effect of the recovery algorithm alone. That is, even the half of the clients that have recovery algorithm disabled still enjoy the other benefits of the architecture. They are able to take advantage of the link-state-based propagation and maintenance of reachability information in the overlay network. They are also able to balance load through the use of the load-balancing algorithm presented in Chapter 5. Take these too into account, the availability improvements presented in Section 6.2.3 are actually an underestimate of the actual availability improvements due to our architecture.

Chapter 7

Conclusions and Future Directions

Service composition enables flexible development of new application functionality through the reuse of service components. This can achieve rapid creation of data services with service mobility features in next-generation networks. In this thesis, we have considered the availability and performance issues in service composition. Our overall approach is to use failure detection and performance information exchange at the service-level; and use alternate service replicas for failure recovery and load balancing. We address several challenges that arise in failure detection, recovery, and load balancing, especially in the presence of scale in different dimensions: number of clients, expanse of the system, and number of services. In this chapter, we summarize the findings and contributions of our work, present concluding discussions, and describe opportunities for further exploration.

7.1 Summary and Concluding Discussions

The issue of availability in wide-area service-level paths arises due to the poor availability of the underlying Internet paths. The problem is compounded by the fact that there is no support in the Internet architecture for quick detection of failures in an inter-domain path. A central theme of this thesis is to see if this issue of failures and their detection can be addressed at the service level.

We start with an analysis of trace data collected over extended periods of time (3-7 days), on a variety of Internet paths, to answer several questions with respect to failure detection. We find the following.

- It is possible to define a notion of failure, or long-lasting outage on an inter-domain Internet path, without relying on the underlying Internet for any support.
- It is possible to detect these long-lasting outages
  - with a heart-beat mechanism with a period of about 300 ms,
  - with a failure detection timeout of about 1,200-1,800 ms, and
  - with a small absolute rate of occurrence of timeouts and false-positives (less than once an hour).

The value of 1,200-1,800 ms for the failure detection timeout thus strikes a good balance between *usefulness* and *feasibility*. It is *useful* since real-time applications can benefit from such a short timeout period to avoid long-lasting outages (several tens of seconds to several minutes). Having such a timeout period is *feasible* since, intuitively, it has a small absolute rate of occurrence: about once an hour or less. Hence the recovery process, and the overhead involved do not occur frequently.

We then design an architecture based on a three-layer model that fits in well with our operational model. The hardware platform of service clusters forms the base on which providers deploy service instances. The use of clusters separates the issue of intra- and inter-cluster failures. It also separates the issue of scaling in the dimension of number of clients/services from the issue of scaling in terms of the expanse of the network: each cluster can be internally incrementally provisioned to accommodate more clients/services.

The service-level overlay network acts as the middleware platform for service composition. It allows exchange of reachability and performance information and provides the context for recovery mechanisms as well as load-balancing algorithms. The overlay network is virtual-circuit based, and this allows for quick recovery independent of the expanse of the system. This is because recovery does not depend on failure information propagating and stabilizing across the network.

We develop an emulation testbed to identify the scaling bottlenecks and overheads in our architecture. We find that the use of a link-state based flooding mechanism to collect entire graph information does not present immediate scaling concerns since the network, memory, and CPU requirements for this are manageable. The virtual-circuit based approach presents scaling concern in the dimension of the number of simultaneous client sessions. However, we find that the additional provisioning required in terms of the cluster managers is minimal in comparison to the provisioning required for the actual service components. Further, within the scaling limits, the time-to-recovery, after failure detection is within 1 second for most client sessions. This means that failure detection and recovery can be done within a few small number of seconds (3-4 seconds).

We examine the issue of load balancing in the context of our architecture. We use the emulation platform for a controlled design study. We introduce the least-inverse-available-capacity (LIAC) metric for choosing service instances for client requests. The interaction between the load balancing metric and the load information propagation mechanism (the feedback loop) is an important aspect of stability in any system. We study the interaction between the LIAC metric and the load information propagation mechanism. We find that a naive approach using periodic load information updates causes load oscillations in time and space (across different service replicas). Increasing the frequency of load updates is not feasible since this brings up an issue of scale – the overhead would be high in a large overlay network.

We introduce a piggybacking mechanism to update load information along the service-level path for a particular client request. While this does not update load globally, it comes with little overhead. Though the load update is only along a path, piggybacking has the property that it reacts to load underestimates quickly. This property helps us achieve minimal load oscillations.

While piggybacking achieves load balancing in space and time, with the LIAC metric, this often means that far-away service instances are chosen even if they are only slightly less loaded. We then introduce the no-op factor in the LIAC metric to avoid choosing far away service instances.

We find through emulation experiments that the LIAC-NF metric combined with the piggybacking mechanism can perform well both in terms of load balancing and service-level path length in a variety of scenarios including single/double link failures.

We illustrate the use of our architecture for service composition using a set of applications in the context of the Universal Inbox. The Universal Inbox achieves extensible any-to-any communication with personal mobility and service mobility in a heterogeneous network setting. It achieves this through name independence, data-type independence, and personalization components. The functional component that achieves data-type independence is the Automatic Path Creation (APC) service that uses composition of transformation and transcoding agents.

We use an implementation of a specific composed application: a text-source composed with a text-to-speech engine, to study our architecture from an application point of view. We deploy the application on a wide-area testbed and have long-running experiments to evaluate the improvement in availability due to the recovery mechanisms. We find that the architecture is able to achieve great improvements in availability through quick failure detection and recovery. The unavailability comes down by factors of up to 2-10 in many cases. For instance, in one case, the unavailability fell from 0.27% to 0.04%, a factor of about seven. In another, the unavailability fell from 0.21% to 0.02%, a factor of over ten, with the use of our recovery algorithm. This is especially significant since the availability is quite good to begin with even without our recovery algorithm, although quite far away from the five 9's availability standards set by telecommunication networks. We are able to push the availability much closer to 100% with our failure detection and recovery mechanisms.

To summarize, the two broad "take-away" points in this thesis are:

- The use of an overlay-level failure detection mechanism – this can be quick (within a couple of seconds), and with a small absolute rate of occurrence of timeouts (about once an hour).
- The use of service-level state to "pin" service-level paths in the overlay network – this allows for session-recovery independent of the size of the network. In contrast, the inter-domain Internet uses datagram-based routing, and a path-vector routing protocol (BGP). In our case, the switching state in the virtual-circuit based routing is used for quick recovery.

The implication of these observations is that quick failure detection and recovery is both feasible and useful for real-time applications. Recovery within a few small number of seconds (3-4 seconds) can mask long-lasting (several tens of seconds to several minutes) failures from the end-user. This significantly reduces system downtime and effectively improves the system availability.

7.2 Directions for Future Work

In the context of service composition there are several aspects that call for further exploration. Our architecture has also brought to the fore many interesting issues that are of interest in a wider context. We describe these now.

7.2.1 Data flow beyond a path

We have considered the case of composition where the data flow is along a service-level path consisting of service components. While this covers a wide range of scenarios in service composition, there are also many others that involve data flow beyond a path. A simple example is the case of a data-flow tree, as shown in Figure 7.1. One can also imagine scenarios in which the data flow graph changes as the session proceeds, with some parts of it lasting longer than other parts. While our generic three-layer model may be applicable in these contexts too, the underlying algorithms and recovery mechanisms are likely to be different and have different trade-offs.

7.2.2 Multiple metrics in service composition

We have considered two important metrics in service composition: reachability and server load. Of these, reachability is boolean as we have defined it. In our load balancing algorithms, we have also considered the hop-count metric. There are also other metrics that one can consider. An obvious network level metric is the available bandwidth on the service-level path. Another metric of interest in a multi-provider setting is the "cost". A user may be subscribed to a particular service provider and may use that provider's instances for composed sessions. However, for better performance, an alternate service provider's instance may be suitable, and may come at a higher cost

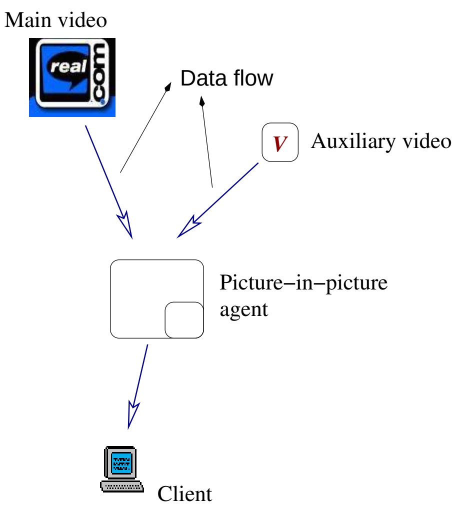

Figure 7.1: Data flow tree: an example

to the user. In such scenarios, there are interesting algorithmic issues of multi-metric optimization as well as system design issues of bringing the user in loop to decide the trade-off between cost and performance dynamically. Consideration of these metrics become especially interesting when the data flow is not just a path.

7.2.3 Dynamic service composition

The operational model under which we have worked is that a third party portal provider decides beforehand as to which kind of services (not the instances) to compose to achieve a particular application functionality. One can imagine situations where the kind of services composed are decided dynamically, on a per-session basis. For instance, to improve perceived performance on a video stream, a transcoder may be dynamically composed in depending on the available bandwidth on the service-level path. The study of the interplay between the introduction of services dynamically into a service-level path and the performance dynamics in the network presents interesting issues. The design of a control loop to achieve this calls for further exploration.

7.2.4 User mobility and dynamic service-level paths

In our architecture, we have studied mechanisms for switching service-level paths on failure detection. There are other scenarios in which one can imagine an alternate service-level path being useful. If a client is mobile, the exit-node for the client may change during the session, and setting up an alternate service-level path may make sense after the user moves. Even otherwise, it may make sense to switch to an alternate path in the middle of a session to achieve better performance. However, such switching may have to be done more frequently than for failure recovery as we have considered in this thesis. This presents issues of stability. It is not clear if overall system performance would improve if each session switches paths independently to improve its performance. It is possible that a set of sessions keep switching back and forth between two service-level paths. It would be interesting to understand the dynamics of a system that uses alternate paths during user mobility or just to improve service-level path performance in the middle of a session.

7.2.5 Stateless paths

In our architecture, the service-level overlay network is virtual-circuit based. That is, the service-level path is pinned, and there is switching state at the overlay nodes. While this helps in quick service-level path recovery by not requiring that failure information propagate and stabilize across the network, it imposes a scaling limit on the number of simultaneous client sessions per cluster manager. This design choice is in line with our goal of quick recovery and availability and also in line with the way service components are written and deployed today (they have per-session state).

An alternative design that is worth exploring is the concept of stateless paths. Here, the overlay nodes do not maintain switching state, but any state required for intermediate processing is carried in the data packets. For instance, if a video codec transformation is required for a video source, this requirement information would be sent in each data packet. The network would treat each packet independently (datagram routing), and would provide the functionality of intelligent intermediate processing in addition to simple routing. This design choice may be more appropriate in setting where quick recovery is not an issue, and where the additional provisioning required to handle per-session state poses a significant overhead (e.g., when service components are light-weight).

The use of stateless paths would necessitate a complete redesign of the mechanisms for failure recovery, load balancing, as well as issues such as service interface matching, or user authentication and security. It would require a definition of an appropriate operational model under which service providers would manage and deploy their instances. This design alternative needs further exploration and evaluation.

7.2.6 Issues related to overlay networks

Our architecture for service composition is based on a service-level overlay network that is built on top of the IP network. This has brought forth a variety of issues related to the deployment and management of overlay networks. Since overlay networks have been proposed for a variety of purposes recently [20, 86, 69, 79], some of these issues apply in a general setting broader than the context of service composition.

There are several related issues with respect to overlay network construction. In the context of our architecture, these are as follows.

- How many overlay nodes are required to span the Internet? We discusses this issue in Section 4.1.4 and argued that for our purposes, a few thousand nodes are probably sufficient. Answering this question precisely requires further exploration. Answering this in a more generic setting is even more challenging. Overlay networks have been proposed for resiliency [20], distributed object location [69, 79, 86], generalized Internet indirection [78], content distribution [2, 32], file storage [51], improving security [50], Internet distance measurement [48], and others. An understanding of the relation between the structure of the underlying Internet and the number of overlay nodes required for these various purposes is a very interesting research issue.

- Where should the overlay nodes be placed? This is an important question related closely to the overlay size. Assuming that we have a choice of overlay node placement, there are trade-offs involved. For instance, placing an overlay node at a particular point in the network may improve fault tolerance by avoiding correlated failures, but may increase the end-to-end latency in service-level paths using that overlay node.

- Where to place service instances in the overlay network? This issue is also closely related to the above two issues. A particular service placement close to the clients may improve the average performance, but may be bad in terms of fault-tolerance – there may not be any alternate service replicas reachable in case of network path failure between the client and a service replica. This trade-off exists in overlay networks in other contexts too. For instance, in an overlay network for content distribution or file storage, having several replicas close together, and close to the clients may improve average performance, but may be bad in terms of fault tolerance.

Another set of issues related to overlay networks in general is that of overlay management. Overlay networks, by definition are independent of the underlying Internet. But they do use the resources of the underlying network. This brings up questions of overlay management, and questions of who is responsible when something goes wrong in terms of poor performance or failure. The problem is even more complicated when the underlying Internet is shared by more than one overlay network. When multiple overlay networks acting independently react to underlying network performance changes or failures, the overall performance may be unpredictable and instability may result. These issues require to be addressed before overlay networks can be adopted as a generic mechanism to solve issues above the Internet layer.

7.2.7 Summary

Composition enables flexible construction of distributed services and is an important technique in next-generation networks. While we have addressed several challenges related to the availability and performance issues in wide-area service composition, we have made some operational assumptions. Distributed service composition can be applied in scenarios wider than we have considered in this thesis, and several related issues of dynamics, performance, and stability arise. We have enlisted a few such broader scenarios and issues in the discussion above. These include consideration of data-flow beyond a path, dynamic composition at various levels, and several open issues related to overlay networks. We hope that our approach taken in this thesis, the design methodology, architecture, and the results form a good platform upon which these broader issues can be addressed effectively in the future.

Bibliography

[1] A Primer on the H.323 Series Standard. http://www.databeam.com/h323/h323primer.html.

[2] Akamai - Delivering a Better Internet. http://www.akamai.com/.

[3] Cisco: LocalDirector. http://www.cisco.com/.

[4] Distributed Component Object Model (DCOM). http://www.microsoft.com/com/tech/DCOM.asp.

[5] i-Mode: DoCoMo's wireless Internet. http://www.eurotechnology.com/imode/faq.html.

[6] Internet2. http://www.internet2.edu/.

[7] Millennium. http://www.millennium.berkeley.edu/.

[8] Nist NET network emulator. http://snad.ncsl.nist.gov/itg/nistnet/.

[9] The now project. http://now.cs.berkeley.edu/.

[10] Object Management Group. http://www.omg.org/.

[11] Open Grid Services Architecture (OGSA). http://www.globus.org/ogsa/.

[12] Open Pluggable Edge Services. http://www.ietf-opes.org/.

[13] The Festival Speech Synthesis System. http://www.cstr.ed.ac.uk/projects/festival.html.

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

[15] The Ninja Project. http://ninja.cs.berkeley.edu.

[16] The R Project for Statistical Computing. http://www.r-project.org/.

[17] A. Acharya and J. Saltz. A Study of Internet Round-Trip Delay. Technical Report CS-TR 3736, UMIACS-TR 96-97, University of Maryland, College Park, 1996-97.

[18] Akamai. Turbo-charging dynamic web sites with Akamai Edgesuite. White paper, 2001.

[19] E. Amir. *An Agent Based Approach to Real-Time Multimedia Transmission over Heterogeneous Environments.* PhD thesis, U.C.Berkeley, 1998.

[20] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *ACM SOSP*, Oct 2001.

[21] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. SWEB: Towards a Scalable World Wide Web Server on Multicomputers. In *IEEE International Symposium on Parallel Processing (IPPS)*, pages 850–856, Apr 1996.

[22] J. G. Apostolopoulos. Reliable video communication over loss packet networks using multiple state encoding and path diversity. In *Visual Communication and Image Processing*, Jan 2001.

[23] M. Baentsch, L. Baum, and G. Molter. Enhancing the Web's Infrastructure: From Caching to Replication. *IEEE Internet Computing*, Mar-Apr 1997.

[24] A. Barbir, E. Burger, R. Chen, S. McHenry, H. Orman, and R. Penno. *OPES Use Cases and Deployment Scenarios, Internet Draft, draft-ietf-opes-scenarios-01.txt*, Aug 2002.

[25] A. Barbir, R. Chen, M. Hofmann, H. Orman, R. Penno, and G. Tomlinson. *An Architecture for Open Pluggable Edge Services (OPES), Internet Draft, draft-ietf-opes-architecture-03.txt*, Aug 2002.

[26] A. Beck and M. Hofmann. *IRML: A Rule Specification Language for Intermediary Services, draft-beck-opes-irml-02.txt*, Nov 2001.

[27] A. Beck, M. Hofmann, and M. Condry. *Example Services for Network Edge Proxies, Internet Draft*, Nov 2000.

[28] J. S. Bendat and A. G. Piersol. *Random Data: Analysis and Measurement Procedures*, chapter 5. Wiley Series, 2000.

[29] A. Bestavros, M. E. Crovella, J. Liu, and D. Martin. Distributed packet rewriting and its application to scalable web server architectures. In *IEEE International Conference on Network Protocols (ICNP)*, Oct 1998.

[30] J. C. Bolot, H. Crepin, and A. V. Garcia. Analysis of audio packet loss in the Internet. In *NOSSDAV*, Apr 1995.

[31] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Zamin. *Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification, RFC: 2205*, Sep 1997.

[32] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed Content Delivery Across Adaptive Overlay Networks. In *ACM SIGCOMM*, Aug 2002.

[33] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing*, May/Jun 1999.

[34] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN Service Availability. In *USITS*, Mar 2001.

[35] T. M. Chen and T. H. Oh. Reliable Services in MPLS. *IEEE Communications Magazine*, Dec 1999.

[36] S. Choi, J. Turner, and T. Wolf. Configuring Sessions in Programmable Networks. In *IEEE INFOCOM*, Apr 2001.

[37] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl*, 2001.

[38] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 25, pages 527–532. McGraw-Hill, 1992.

[39] J. Degener and C. Bormann. GSM Codec Library, University TU-Berlin, FB-Informatik. http://kbs.cs.tu-berlin.de/ jutta/toast.html.

[40] A. Fox. *A Framework for Separating Server Scalability and Availability from Internet Application Functionality.* PhD thesis, U.C.Berkeley, 1998.

[41] A. Fox and E. A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.

[42] M. Fry and A. Ghosh. Application level active networking. *Computer Networks*, 31(7):655–667, 1999.

[43] A. Ghosh, M. Fry, and J. Crowcroft. An Architecture for Application Layer Routing. In *IWAN*, Oct 2000.

[44] S. Gribble and et.al. Scalable, Distributed Data Structures for Internet Service Construction. In *OSDI'00*, Oct 2000.

[45] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. *Computer Networks and ISDN Systems*, 1998.

[46] David S. Isenberg. The Dawn of the Stupid Network. *ACM Networker 2.1*, pages 24–31, Feb/Mar 1998.

[47] V. Jacobson and S. McCanne. VAT Mbone Audio Conferencing Software. ftp://ftp.ee.lbl.gov/conferencing/vat.

[48] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. On the Placement of Internet Instrumentation. In *IEEE INFOCOM*, Mar 2000.

[49] A. D. Joseph, B. Hohlt, R. H. Katz, and E. Kiciman. System Support for Multimodal Information Access and Device Control. Work in Progress, Workshop on Mobile Computing Systems and Applications (WMCSA), 1999.

[50] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *ACM SIGCOMM*, Aug 2002.

[51] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, Nov 2000.

[52] D. R. Kuhn. Sources of Failure in the Public Switched Telephone Network. *IEEE Computer*, Apr 1997.

[53] T. T. Kwan, R. E. McGrath, and D. A. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer*, Nov 1995.

[54] C. Labovitz, A. Ahuja, A. Abose, and F. Jahanian. An Experimental Study of Delayed Internet Routing Convergence. In *ACM SIGCOMM*, Aug/Sep 2000.

[55] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental Study of Internet Stability and Wide-Area Network Failures. In *FTCS*, 1999.

[56] F. Leymann. *Web Services Flow Language (WSFL 1.0), http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf*, May 2001.

[57] G. Ljung and G. Box. On a Measure of Lack of Fit in Time Series Models. *Biometrika*, pages 553–564, 1978.

[58] Q. Ma and P. Steenkiste. On Path Selection for Traffic with Bandwidth Guarantees. In *ICNP*, Oct 1997.

[59] W. Mohr and W. Konhauser. Access Network Evolution Beyond Third Generation Mobile Communications. *IEEE Communications Magazine*, Dec 2000.

[60] M. Mouly and M. B. Pautet. *The GSM System for Mobile Communications*, chapter 2, pages 102,133. Cell & Sys, 1992.

[61] T. J. Mowbray and R. Zahavi. *The Essential CORBA: System Integration Using Distributed Objects*. Wiley Computer Pub., 1997.

[62] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, 1981.

[63] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *ASPLOS*, Oct 1998.

[64] R. Pandya. Emerging Mobile and Personal Communiction Systems. *IEEE Communications Magazine*, Jun 1995.

[65] R. Pandya, D. Grillo, E. Lycksell, P. Mieybegue, H. Okinaka, and M. Yabusaki. IMT-2000 standards: network aspects. *IEEE Personal Communications Magazine*, Aug 1997.

[66] M. Rahnema. Overview of The GSM System and Protocol Architecture. *IEEE Communications Magazine*, Apr 1993.

[67] B. Raman, R. H. Katz, and A. D. Joseph. Universal Inbox: Providing Extensible Personal Mobility and Service Mobility in an Integrated Communication Network. In *WMCSA*, Dec 2000.

[68] J. Rapeli. UMTS: targets, system concept, and standardization in a global framework. *IEEE Personal Communications Magazine*, Feb 1995.

[69] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *ACM SIGCOMM*, Aug 2001.

[70] Y. Rekhter and T. Li. *A Border Gateway Protocol 4 (BGP-4)*, Mar 1995.

[71] M. Roussopoulos and et.al. Personal-level Routing in the Mobile People Architecture. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Oct 1999.

[72] J. Sametinger. A Taxonomy for Software Components. In *Workshop on Component-Oriented Programming (WCOP)*, Jul 1996.

[73] J. Sametinger. Classification of Composition and Interoperation. In *Poster Session, OOPSLA*, Oct 1996.

[74] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. E. Anderson. The End-to-End Effects of Internet Path Selection. In *ACM SIGCOMM*, pages 289–299, Aug/Sep 1999.

[75] H. Schulzrinne. A Comprehensive Multimedia Control Architecture for the Internet. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, May 1997.

[76] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. In *USITS*, Dec 1997.

[77] A. Singhai, S. B. Lim, and S. R. Radia. The SunSCALR Framework for Internet Servers. In *IEEE Fault-Tolerant Computing Systems*, Jun 1998.

[78] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *ACM SIGCOMM*, Aug 2002.

[79] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, pages 149–160, Aug 2001.

[80] S. J. Vaughan-Nichols. Web Services: Beyond the Hype. *IEEE Computer*, Feb 2002.

[81] H. J. Wang, B. Raman, C-N. Chuah, R. Biswas, R. Gummadi, B. Hohlt, X. Hong, E. Kiciman, Z. Mao, J. Shih, L. Subramanian, B. Y. Zhao, A. D. Joseph, and R. H. Katz. ICEBERG: An Internet-core Network Architecture for Integrated Communications. *IEEE Personal Communications Magazine*, Aug 2000.

[82] M. Yajnik, S. B. Moon, J. F. Kurose, and D. F. Towsley. Measurement and Modeling of Temporal Dependence in Packet Loss. In *IEEE INFOCOM*, Mar 1999.

[83] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Usenix*, Jan 1997.

[84] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *IEEE INFOCOM*, Apr 1996.

[85] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the Constancy of Internet Path Properties. In *ACM SIGCOMM Internet Measurement Workshop*, Nov 2001.

[86] B. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, 2001.

Appendix A

Failure Detection Timeout Events: A Time-Series Analysis

In our analysis of the trace-data in Section 3.1.2, we did not consider the time-correlation of the data. If the timeout events, for a particular value of the timeout can be shown to exhibit time correlations, then it is conceivable that an adaptive timeout mechanism can be designed, rather than choosing a fixed timeout. For instance, if timeouts are known to be correlated within, say 5-minute periods, then the timeout value can be slowly increased on detecting frequent timeouts within a 5-minute period. An alternative possibility is to avoid, or at least discourage the use of an overlay link on which a failure has been detected in the recent past. We now consider this aspect of the analysis through statistical methods.

Packet-loss events have been analyzed and shown to be correlated within 1-second periods in [82, 30]. In [82], up to a few hours of traces are considered for analysis. In our scenario, we are interested in the distribution of timeout events (and not packet-loss events), and the time-scale of operation is quite different than in [82, 30] (several days, as opposed to a few hours of data). In [85], an analysis of the time-correlation of loss episodes of 20 Hz measurements (20 packets per second on an Internet path) is performed. The loss episodes are similar to the timeout events in our scenario in that they capture a train of losses. However, again, the time-scale considered in [85] is quite different. It analyzes hour-long traces, and considers time-correlations within about 5-10 seconds. On the contrary, we are interested in (a) timeouts, which by themselves represent outages ranging from 1.2 seconds to several tens of seconds, (b) time-correlation at larger time-scales (say, a few minutes or more – since a single outage could itself last for this long), and (c) consider traces that run for several days.

We use the empirical value of 1,800 ms for the timeout value in our analysis below (recall that the knees in the plots of failure occurrence probability distribution in Chapter 3 occur around 1,200-1,800 ms). We identify the timeout events and consider it as a binary-valued time-series. For every heart-beat period (300 ms), we define the time-series variable value to be 1 if it is the beginning of a timeout event, 0 otherwise. That is:

$$\begin{aligned} Y_i &= 1, \text{ if the } i^{th} \text{ heart-beat is the beginning of a timeout event} \\ & 0, \text{ otherwise} \end{aligned}$$

We thus derive $\{Y_i\}$ directly from the heart-beat trace data. Since we collected data over several days, the above definition of a time-series yields a very large data set (288,000 data points per day). For ease of analysis, we make a slight modification to the data as follows. Instead of considering 300 ms periods, we consider 3 second periods and define another time-series variable as:

$$\begin{aligned} Z_i &= 1, \text{ if the } i^{th} \text{ 3-second period is the beginning of a timeout event} \\ & 0, \text{ otherwise} \end{aligned}$$

This effectively reduces the data size by a factor of ten, and suffices for the purposes of studying time-correlation in the data set. Our analysis proceeds as follows. We assume stationarity of the distribution of the failure events. We first plot the inter-arrival times of failure events in a time-series. This suggests that failure events may be correlated in time. We then plot the auto-correlation function at various lags, as well as the spectral density plots of the time-series of failure events. These plots further indicate possibilities of time-correlation in several of the traces. We then use the Box-Ljung Q-statistic [57] (also used in [85]) to analyze time-correlations in the $\{Z_i\}$ time-series. Our statistical test is against the null hypothesis that the timeout events are IID. We find that the hypothesis is rejected even at large time-lags (5 minutes).

Inter-arrival plots

Ideally, a time-series analysis begins with a time-series plot to observe the data visually. However, in our case, given the very small number of failure events spaced over a very large duration, we instead plot the inter-arrival times of the failures. Figures A.1, A.2, and A.3 show the timeout event inter-arrival times for the eighteen traces. Note that the x-axis represents the failure event count, and not time on an absolute scale. The y-axis is in log-scale, since the failure inter-arrival times have a wide range. We look for time-correlations in time-scales of a few minutes (100-1000 seconds).

We find that in several traces (3, 5, 6, 8, 9, 10, 12, 13, 14, 17), there is a long sequence of timeout events closely spaced in time. In many of these cases, these timeout events are spaced further than the 5-10 seconds as considered in [85]. For instance, in trace #10, there is a sequence of timeout events spaced apart by 50-100 seconds or more.

This suggests that the timeouts that are closely spaced in time are probably due to high network congestion, as opposed to other causes of failure or BGP route change. To quantify the time-correlation seen, we compute the auto-correlation functions next.

Auto-correlation plots

The auto-correlation of a time-series $\{Z_i\}$ at lag h is the estimated correlation between the values Z_i and Z_{i+h} [28]. Figures A.4, A.5, and A.6 show the auto-correlation function plotted against the lag for the eighteen different traces. We consider lags of up to a 100 in these plots. Since in our time-series $\{Z_i\}$, the readings are separated by 3 seconds, this corresponds to a maximum lag of 300 seconds, or 5 minutes.

The auto-correlation plots show that for the same traces in which we saw regions of closely spaced timeout events in the previous set of inter-arrival plots, we now see “significant” auto-correlations for various values of lags (statistical significance of these auto-correlations still needs to be tested). We noted earlier that time-correlation in the timeout events suggests that the timeouts are due to network congestion, as opposed to failures due to other causes. In many of the auto-correlation functions, we notice that correlation dies out beyond a specific time-lag. This is particularly true for traces no: 3, 6, 8, 10, 12, and 17. This means that the network congestion on these Internet paths lasts for a particular time period (the same as the time-lag up to which there is auto-correlation). That is, there is a burst of traffic that lasts for this period of time – which varies from a minute to about three minutes for the various plots. This suggests that traffic engineering solutions that operate within a network domain on a per-minute time-scale will probably be useful. (From our traces, we could not answer whether these were actually in use on the Internet paths we studied).

We next look at the data in the frequency domain by plotting the estimated spectral density functions.

Spectral density plots

The spectral density plots in Figures A.7, A.8, and A.9 are from a fast fourier transform of the time-series. The x-axis shows the frequency, with a maximum value of half, which corresponds to adjacent values in the time-series. The y-axis shows the spectral density [28] on a log-scale. The estimation uses a 95% confidence interval. The periodograms are smoothed with a series of modified Daniell smoothers (using the R statistics package [16]). A flat spectral density function results from white noise (independent, random data), while peaks in the plot mean that there is periodicity in the data at that particular frequency. For instance, if there is a peak at a frequency of about 0.33, this would mean that the data shows a positive correlation at a frequency of once in three data points.

The spectral density plots reinforce the observations made using the previous plots. Many of the traces show time-correlation in the data at various frequencies. (For traces no: 16 and 18, the time-correlation is extremely low, and the spectral density estimates fall below the chosen scale on the y-axis).

Statistical test of time-correlation

We use the Box-Ljung Q-statistic [57] to test for time-correlation in the failure event time-series represented by $\{Z_i\}$. (The same test is used in [85] as well). The Q statistic tests for near-term correlations *up to* a given lag k in the time-series. For a given time-series of n elements, and a given lag k, the Q-statistic is defined as the weighted sum of squares of the auto-correlations from lag 1 to k. That is:

$$Q_k \;=\; n \times (n+2) \sum_{i=1}^{k} \frac{r_i^2}{(n-i)}$$

Here, r_i is the auto-correlation of the time-series at lag i. When n is large, under the null hypothesis that the time-series is white noise, Q_k has approximately the χ^2 distribution with k degrees of freedom. We can thus compare the Q_k with the known χ^2 distribution under a desired significance level to test the hypothesis. Table A.1 summarizes the results of the test for the eighteen traces. We are interested in looking for correlation within a few minutes – we choose k accordingly. We perform the statistical test for two values of k – 50 and 100. These correspond to time-lags of $50 \times 3 = 150$ seconds (2.5 minutes) and $100 \times 3 = 300$ seconds (5 minutes). We see that the hypothesis is rejected quite strongly at the 95% significance level at lags of up to 50 as well as up to 100 for most of the traces. We also note that the cases where we are not able to reject the hypothesis are the cases where there are very few number of failures. This suggests that the timeouts that happen due to reasons other than network congestion are independent of one another, without any time-correlation.

Summary

We conclude from this analysis that there are time-correlations in the timeout events. The likely cause for this is network congestion. The time-scale of correlation is much greater than that known for packet-loss events (within 1 second [82, 30]). What this means with respect to our architecture is that we can probably design an adaptive timeout mechanism to further reduce the absolute rate of occurrence of timeouts. Alternatively, the architecture can use these time-correlations to discourage the use of a particular overlay link for a period of time. These possibilities

Trace no:	HB Destn.	HB Src.	Number of Failures	Q_{50}	Q_{100}	Result at 95% sig. Q_{50}, Q_{100}
1	UNSW	Berkeley	18	2886.3	2886.4	reject, reject
2	UNSW	TU-Berlin	28	2190.7	2190.95	reject, reject
3	CMU	Berkeley	47	197848	199106	reject, reject
4	CMU	Stanford	14	3084.57	6169.75	reject, reject
5	Berkeley	UNSW	187	32914.5	53834.4	reject, reject
6	Berkeley	CMU	87	99151.2	124125	reject, reject
7	Berkeley	UIUC	12	3103.27	3103.35	reject, reject
8	Berkeley	TU-Berlin	34	21604.6	21605	reject, reject
9	Berkeley	Stanford	263	26679.5	33032.9	reject, reject
10	Berkeley(2)	Stanford(2)	72	128786	152785	reject, reject
11	UIUC	Berkeley	9	0.0452319	0.0905396	non-reject, non-reject
12	UIUC	Stanford	74	13551.4	17053.7	reject, reject
13	TU-Berlin	UNSW	127	47878.6	78191.7	reject, reject
14	TU-Berlin	Berkeley	164	153783	228031	reject, reject
15	Stanford	CMU	20	2011.33	4023.13	reject, reject
16	Stanford	Berkeley	9	0.0273614	0.0547506	non-reject, non-reject
17	Stanford(2)	Berkeley(2)	41	398399	403651	reject, reject
18	Stanford	UIUC	5	0.0138139	3624.35	non-reject, reject

Table A.1: Q-statistic test for time-correlation; $\chi^2(0.95, 50) = 67.5$, and $\chi^2(0.95, 100) = 127.3$

call for further design and evaluation.

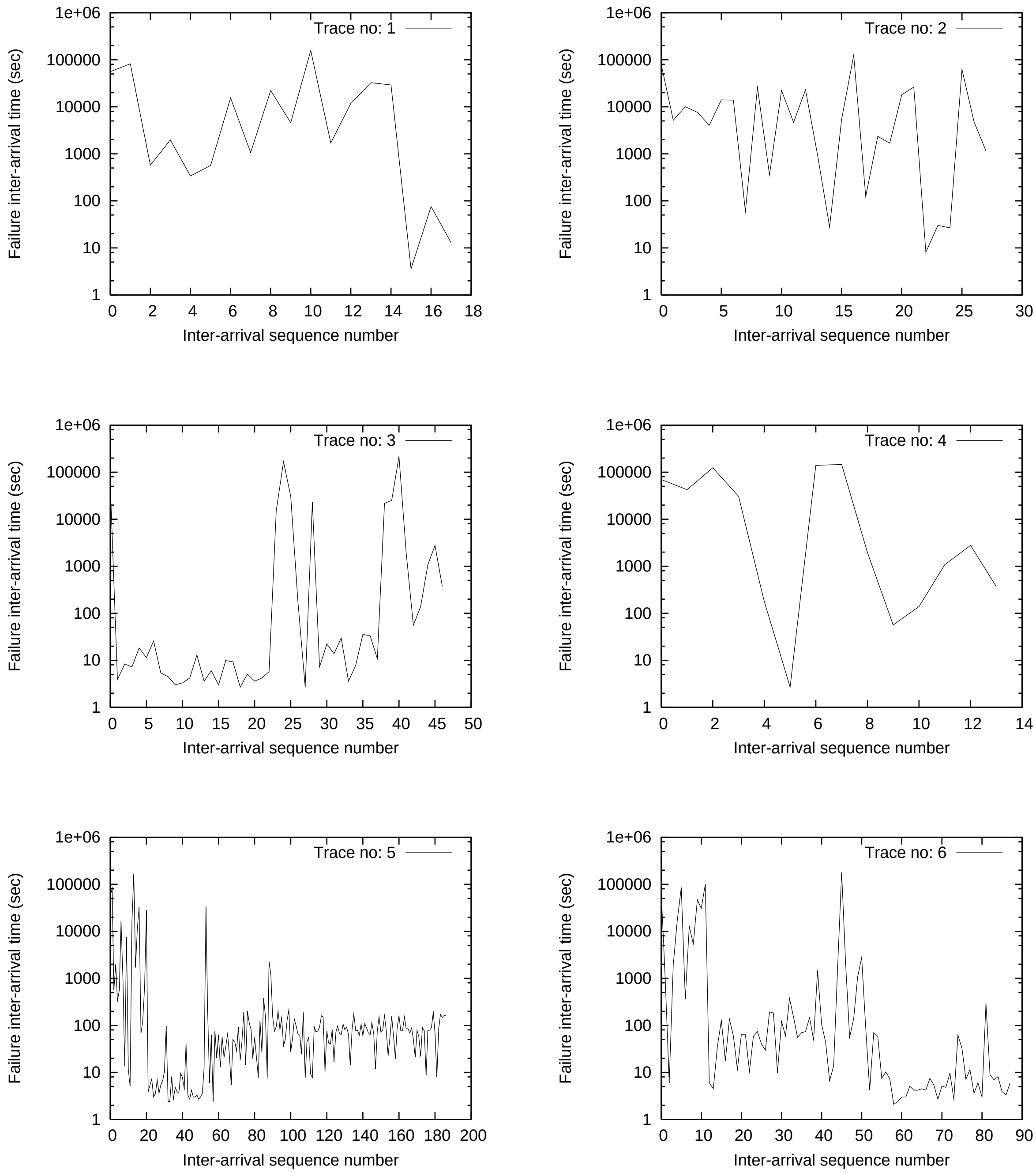

Figure A.1: Failure inter-arrival plot, Traces 1-6

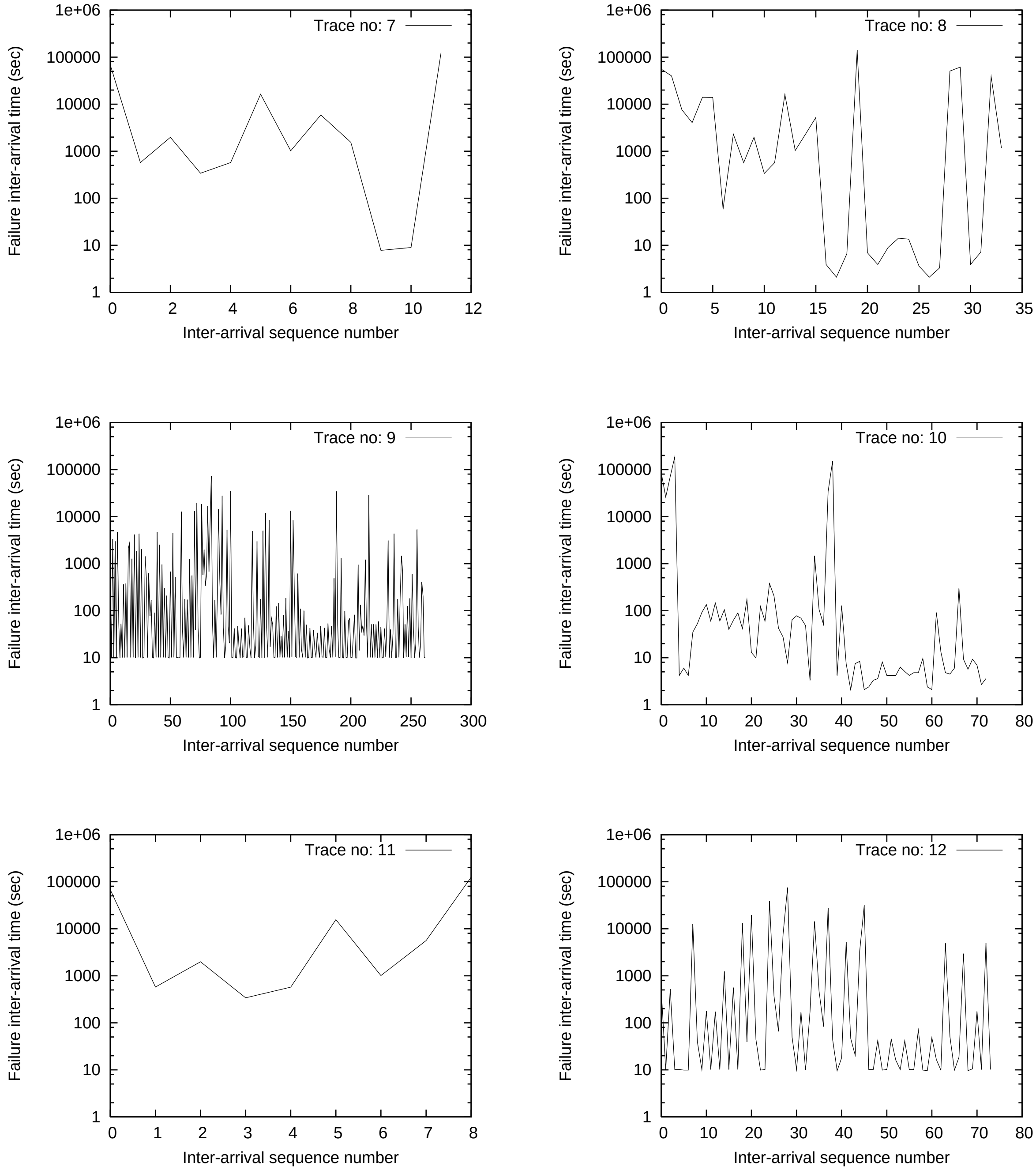

Figure A.2: Failure inter-arrival plot, Traces 7-12

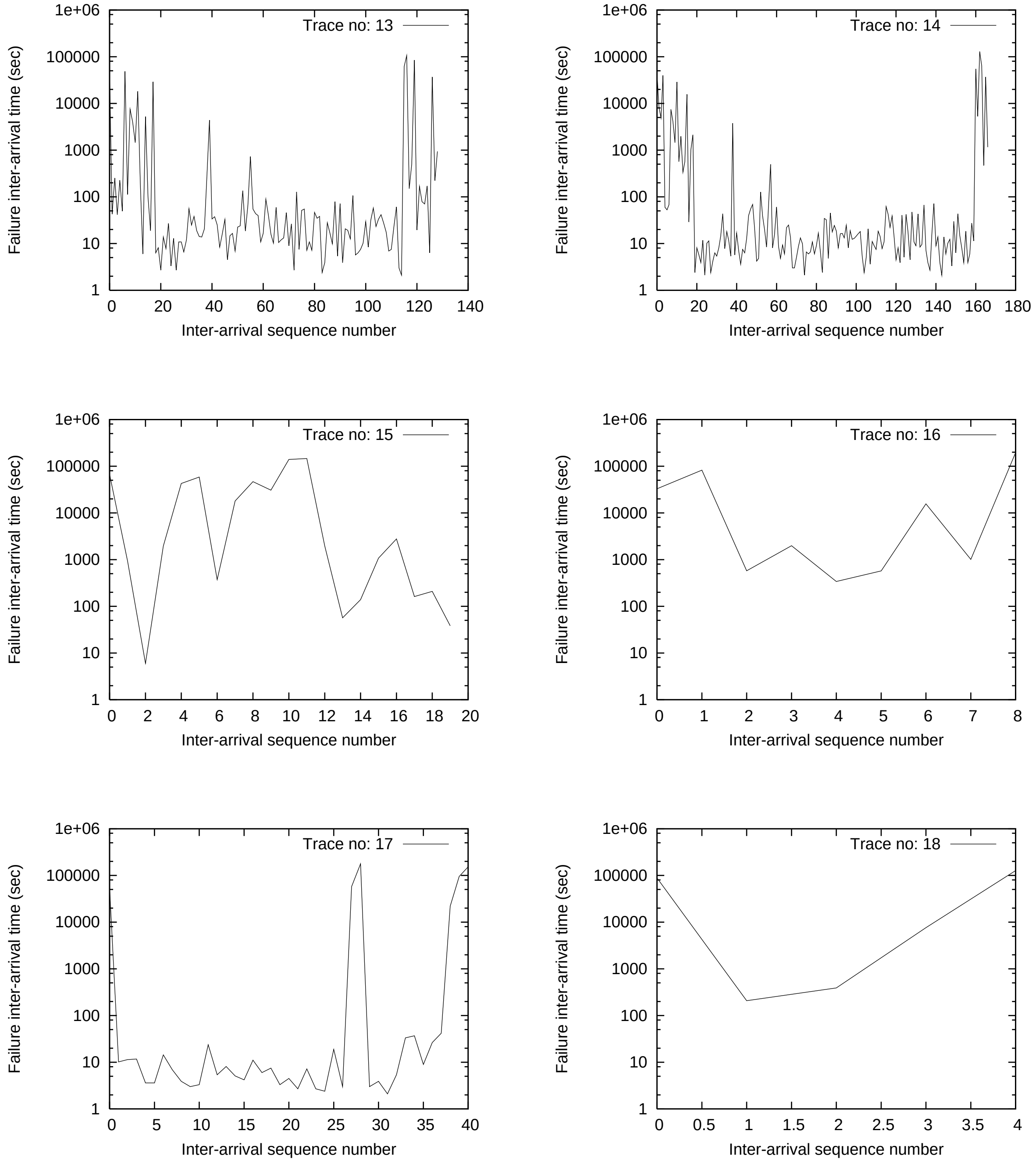

Figure A.3: Failure inter-arrival plot, Traces 13-18

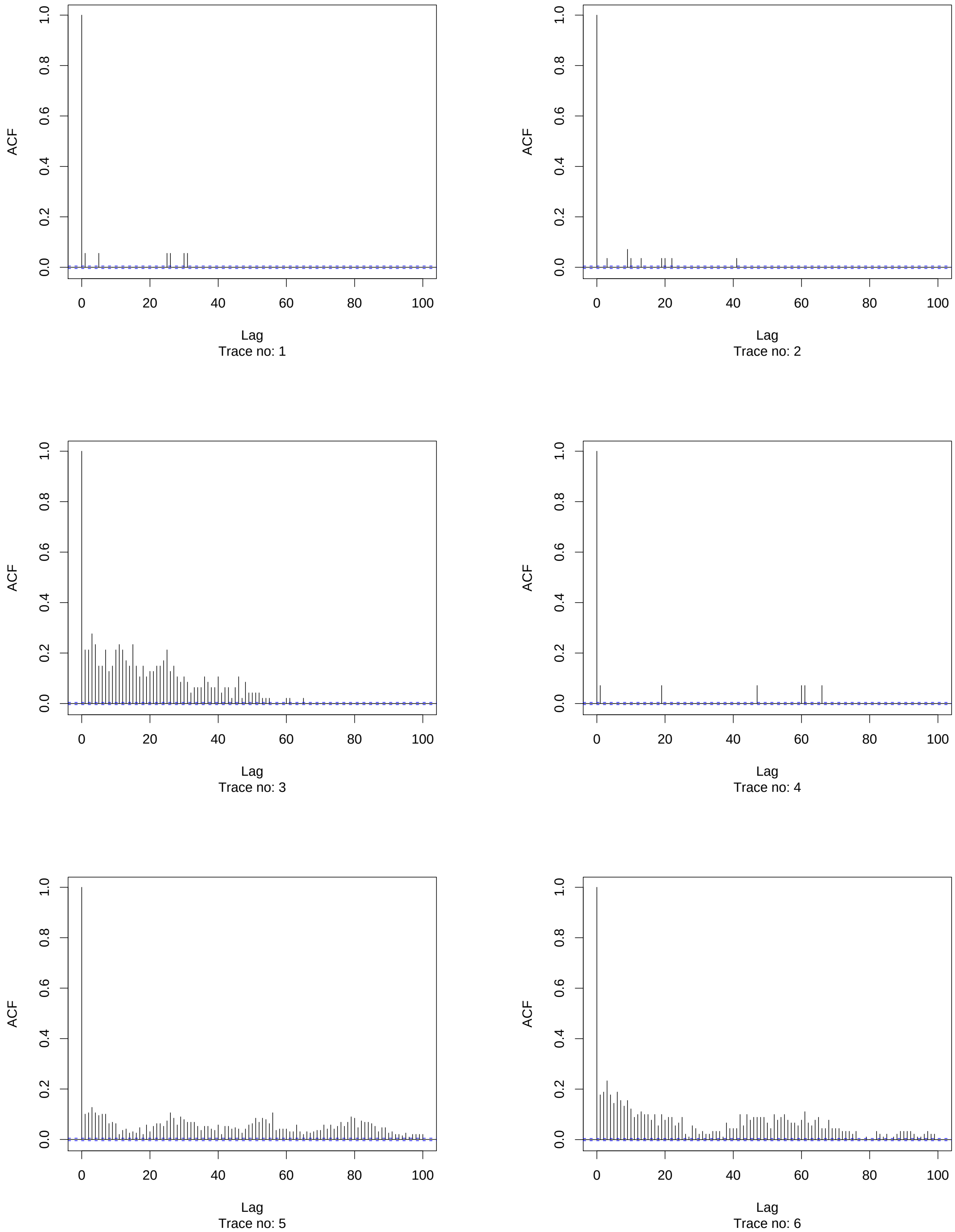

Figure A.4: Auto-correlation function, Traces 1-6

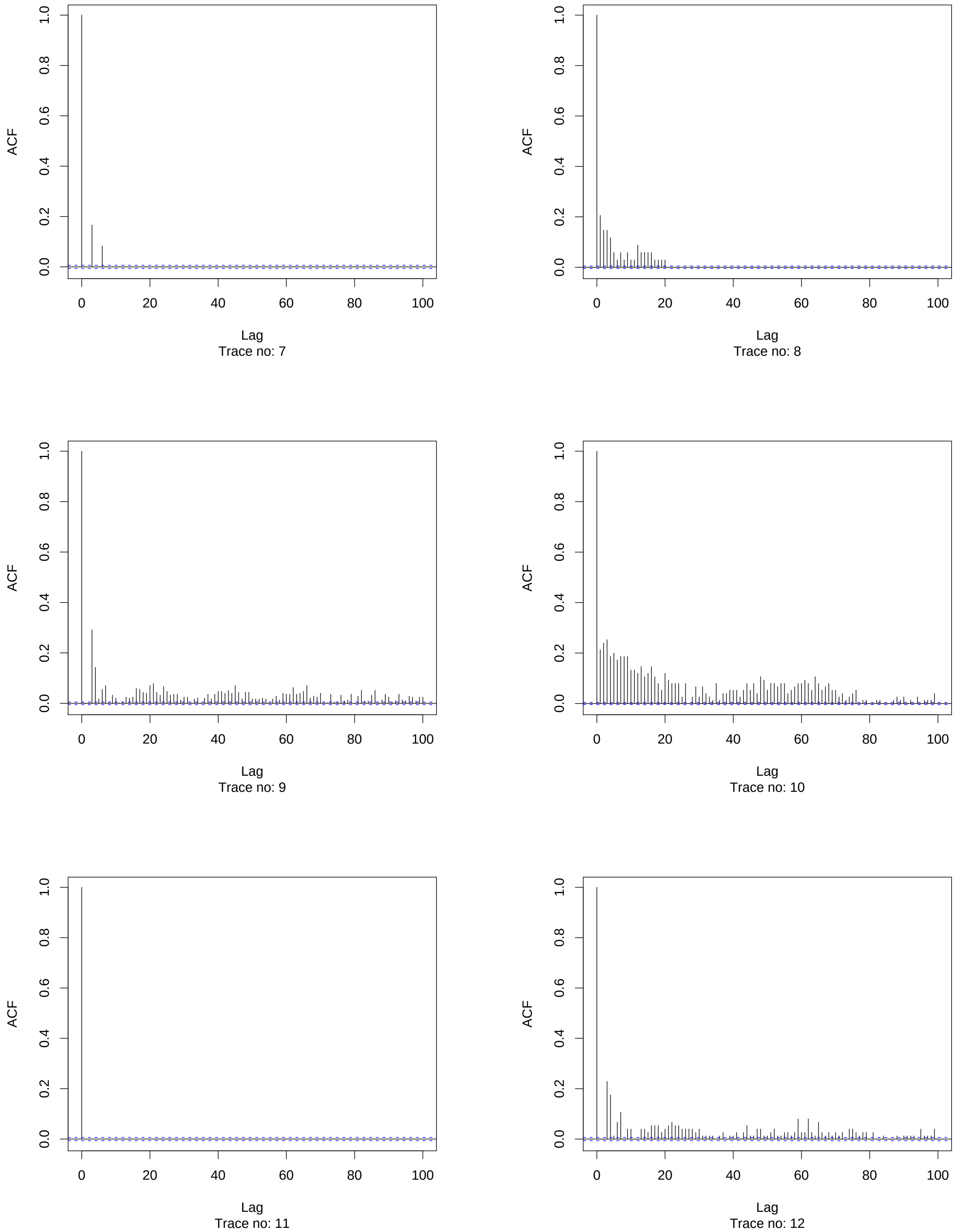

Figure A.5: Auto-correlation function, Traces 7-12

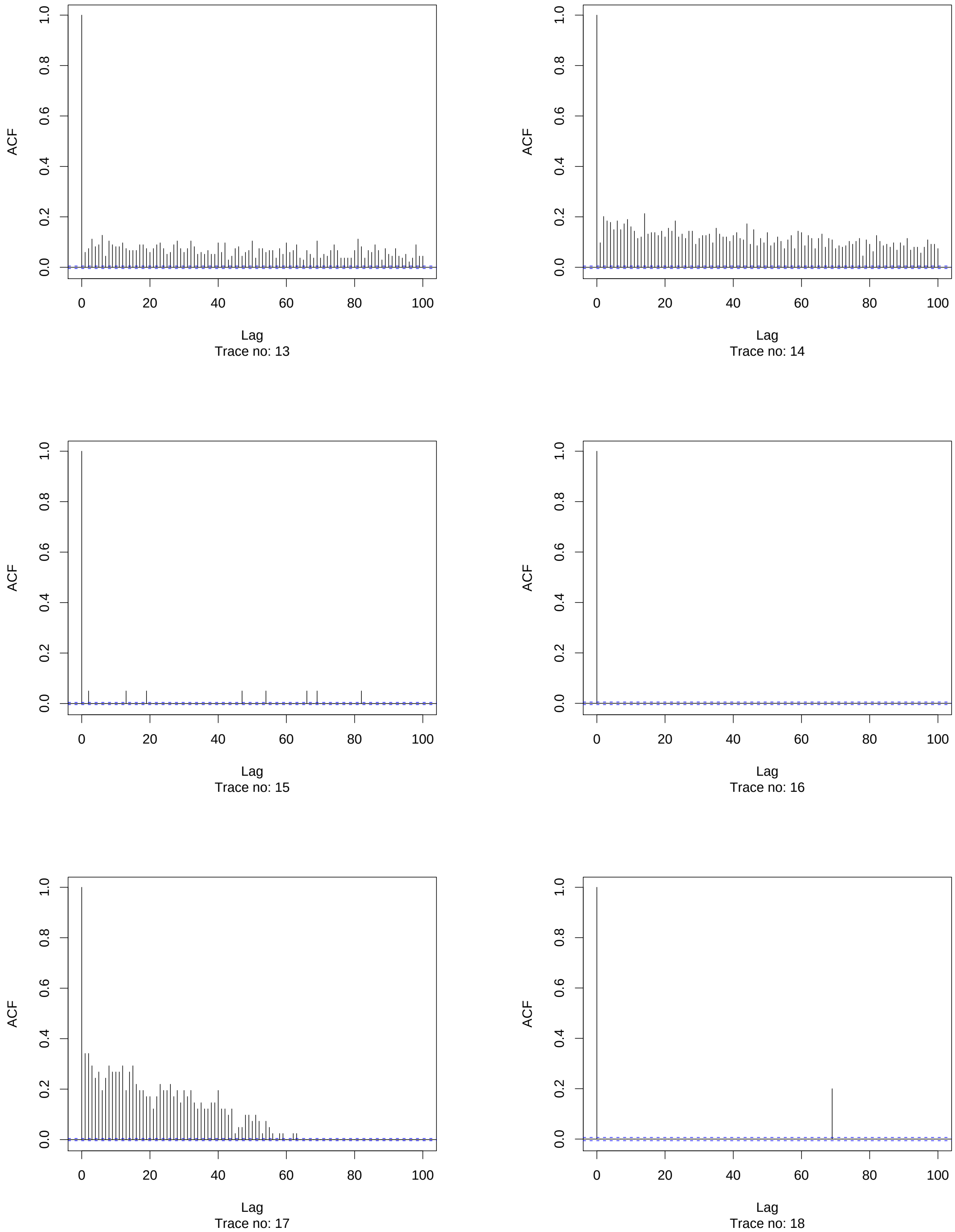

Figure A.6: Auto-correlation function, Traces 13-18

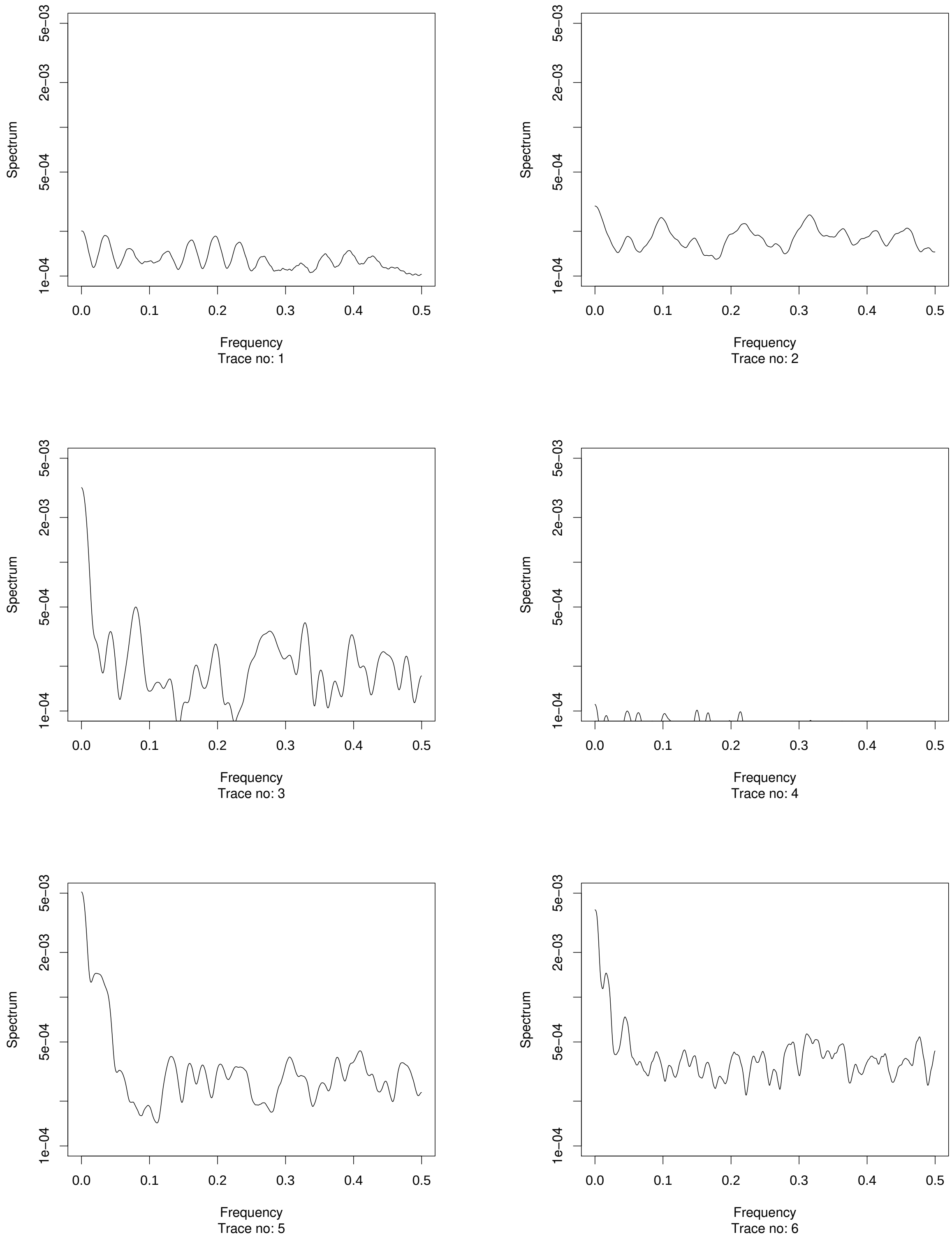

Figure A.7: Spectral density function, Traces 1-6

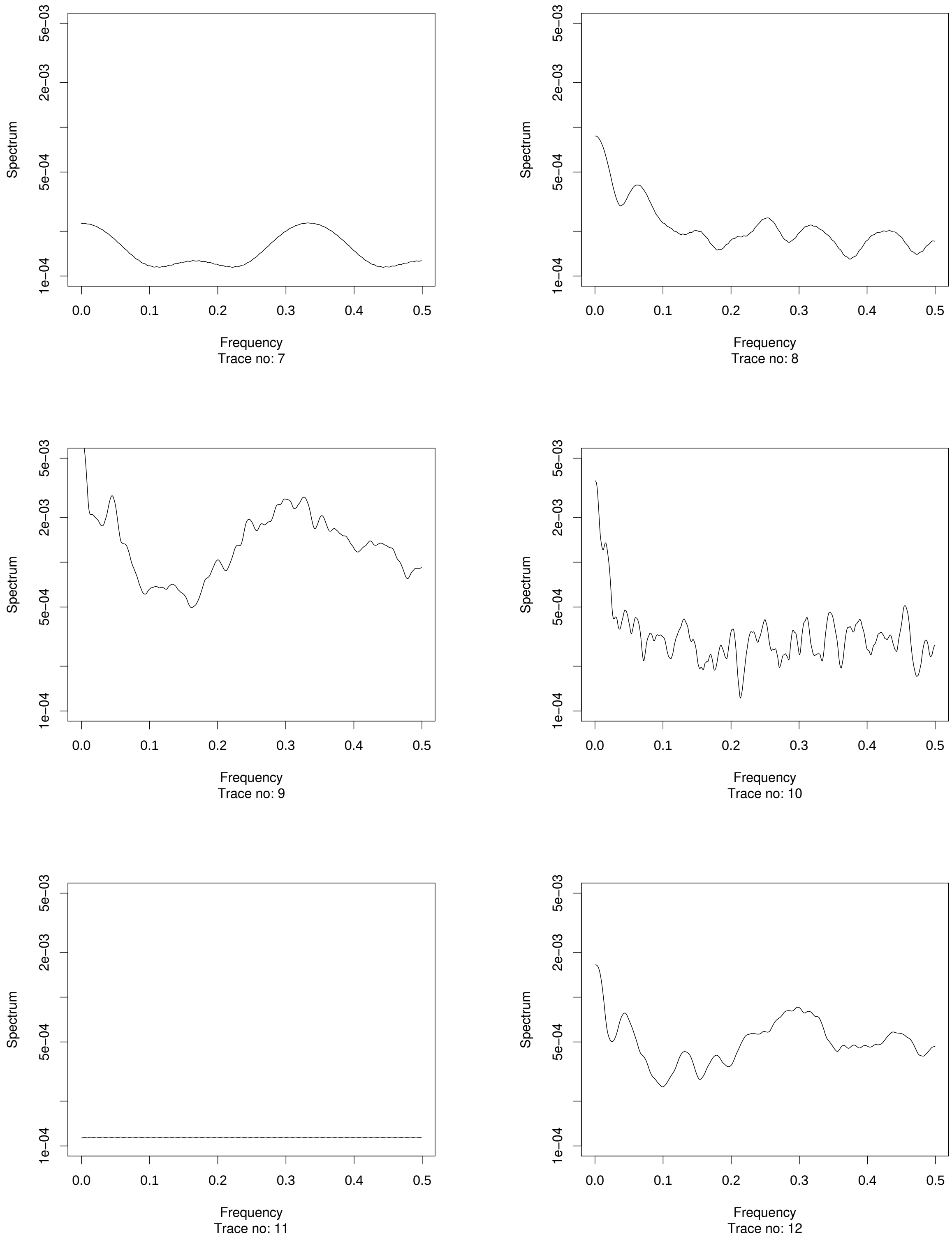

Figure A.8: Spectral density function, Traces 7-12

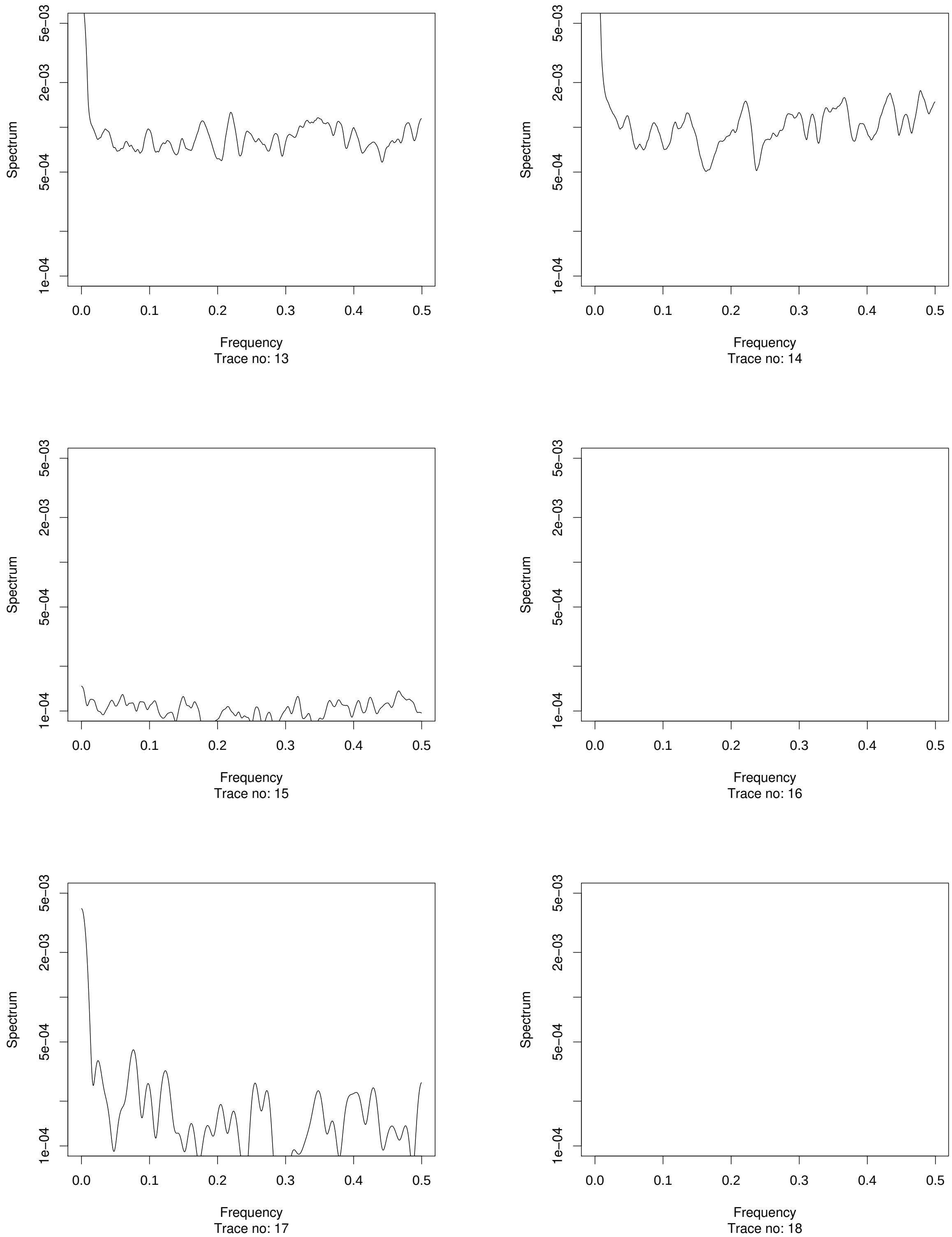

Figure A.9: Spectral density function, Traces 13-18