Emulation-based Evaluation of an Architecture for Wide-Area Service Composition

Bhaskaran Raman **Randy H. Katz**

475 Soda Hall, EECS Department, U.C.Berkeley
Berkeley, CA 94720-1776, U.S.A.
{bhaskar,randy}@cs.berkeley.edu

Abstract

Service composition provides a flexible way to quickly enable new application functionalities using component services. We focus on the scenario where next generation portal providers "compose" the services of other providers. We have developed an architecture based on an overlay network of service clusters to provide failure-resilient composition of services across the wide-area Internet: our algorithms detect and recover quickly from failures in composed client sessions.

In this paper, we present an evaluation of our architecture whose overarching goal is quick recovery of client sessions. The evaluation of an Internet-scale system like ours is challenging. Simulations do not capture true workload conditions and Internet-wide deployments are often infeasible. We have developed an emulation platform for our evaluation – one that allows a realistic and controlled design study. Our experiments show the effectiveness of our recovery mechanisms: over 90% of the client sessions are restored within 1sec after failure detection in Internet paths. We collect trace data to show that failure detection itself can be tight on wide-area Internet paths – within about 2sec. Failure detection and recovery within these time bounds represents a significant improvement over existing Internet path recovery mechanisms that take several tens of seconds to a few minutes [12]. Furthermore, the control overhead involved in implementing our recovery mechanism is minimal in terms of network as well as processor resources; minimal additional provisioning is required for this.

Keywords: Network emulation, Service composition, Overlay networks, Failure detection, Session recovery

1 Introduction

Value added services and content provisioning will be the driving force behind the development and deployment of future communication networks. It is important to enable quick and flexible development of end application functionality. Composition of services from independent components offers a flexible way to enable new application functionalities. Consider for instance a user with a new wireless thin client roaming to a foreign network. She 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 (Fig. 1). Further, 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 third-party text-to-speech conversion engine, with the user's email repository. Composition of complex services from primitive components enables quick development of new application functionality through the reuse of the components for multiple compositions.

In a composed service, a set of component services are strung together – we call this a *service-level path*.

Composition by itself is not a novel idea. However, there are critical challenges to be addressed in the context of composing independent components across multiple service providers. In our example, the transcoding service and the news/weather video service could belong to different providers. Providers deploy and manage multiple instances of their services at different points in the network for load balancing and availability reasons. The *performance* of the composed service is critically dependent on the choice of instances for composition. This is more challenging than traditional web-server selection since we have to choose a *set* of service instances, with adequate network performance along the entire path. These choices have to be dynamic, and on a per-client basis, taking into account the client's position in the network.

A second challenge is that of *availability*. Since services are deployed by multiple providers, a service-level path could span multiple network domains (Fig. 1). This has implications on the availability of the composed service. Recent research has shown that inter-domain Internet path availability is very poor [13], and that Internet route recovery can take of the order of a few minutes [12]. Since multimedia sessions could last for several minutes to hours, it is important to address network failures *during* a session. Such recovery has to be *quick* for real-time applications. We wish to take advantage of the multiple replicas of the service instances and dynamically choose alternate service instances when the original service-level path experiences an outage[1] (see the dotted lines in Fig. 1).

Quick restoration of service-level paths is challenging since there are *scaling* implications when a large number of client sessions have to be restored on a failure. Another challenge is that

[1]We assume that services have only soft-state, and no persistent state. Soft-state can be built up at an alternate server without affecting the correctness of the session. Our examples fall under this category. Also see [1].

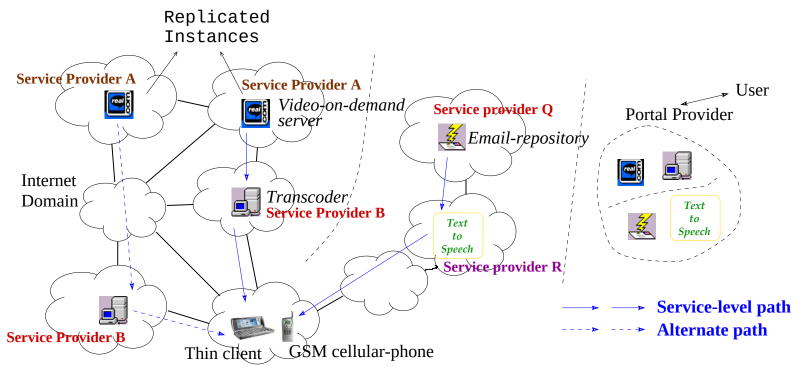


Figure 1. Service composition across the wide-area Internet

of *failure detection* over the wide-area Internet. There is inherent variability in delay, loss-rates, and outage durations. A conservative timeout mechanism to detect failures could mean longer detection times in general, while a more aggressive mechanism may trigger *spurious* path restorations.

We have developed an architecture for addressing the issues of performance and availability in service-level paths. In this paper, we present an evaluation of our design. We specifically look at the issue of quick failure detection and recovery for availability of the service-level path. A challenge that relates to evaluation of mechanisms for path recovery is the following. Simulations are not ideal for capturing true processor/network overheads, especially under scale. However, creating and maintaining a realistic research testbed across the wide-area Internet would be too cumbersome and expensive. Also, with a real deployment, a controlled design study would be difficult due to non-repeatability of experimental conditions.

In this paper, we present an evaluation of our path-recovery algorithms using an emulation platform that we have developed. The platform allows us to realistically implement our algorithms, while emulating wide-area Internet latency and loss. The different instances of our distributed recovery algorithm run on multiple machines of a cluster within our testbed. Unlike a simulation, the emulation can run in real-time, handling the control traffic of thousands of client sessions; and unlike a real deployment, the testbed itself, and the experimental conditions are under our control.

Our experiments show that the control overhead involved in updating the distributed path state to effect restoration is manageable, both in terms of network resources as well as processor resources. This allows the system to scale well with an increasing number of simultaneous client sessions. In our implementation, a single machine (Pentium-III 500MHz) can easily handle the distributed path state associated with about 400-500 simultaneous client sessions. (Beyond this, we run into bottlenecks in our emulation setup). This amounts to little additional provisioning, especially when dealing with heavy-weight service components such as the video transcoder or the text-to-speech engine in our example.

To analyze how quickly failure *detection* can be done, we collect trace-data on Internet path outages across geographically distributed hosts. Our analysis shows that failure detection itself, over the wide-area can be done quite aggressively, within about 2sec. We use the traces to model losses and outages on Internet paths and use this to drive our emulation. We find that even with an aggressive failure detection timeout of 1.8sec, spurious path restorations happen infrequently – about once an hour. Since the control overhead is small, such spurious restorations are a small price to pay for quick failure detection.

Further, under our trace-based modeling of Internet outages, we find that *recovery* of paths after failure detection can be done within 1sec for over 90% of the client sessions. Such a combination of quick detection and recovery, within a few small number of seconds, would be immensely useful for the kinds of real-time applications described above.

The next section presents an overview of our architecture. Section 3 describes the emulation testbed. Our evaluation of path recovery mechanisms is in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Design Overview

In this section, we present a brief walk-through of our architecture, highlighting the main design points to establish the context for the performance evaluation. The goal of our architecture is to enable performance sensitive service-level path creation, and path recovery upon failure detection. The idea behind path recovery is to use an alternate Internet path, much as in [4, 5]. The motivation for this is that network-level failures can happen quite often – studies show that inter-domain Internet paths can have availability as low as 95% [13]. And when such failures do happen, they can last for several minutes [12].

The choice of service instances for service-level path creation/recovery is somewhat like web-mirror selection, but is more complicated, since in general, we may need to select a *set* of instances for a client session. Further, 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.

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.

Motivated by the notion of a service-level path, we think in terms of a service-level *overlay network*. Our architecture for optimal and robust service composition is depicted in Fig. 2. We have three planes of operation: at the lowest layer is the hardware platform consisting of compute clusters deployed at multiple points on the Internet. This constitutes the middle-ware platform on which service providers deploy services. Providers could deploy their own service clusters, or could use third party providers' clusters. We define a logical overlay network on top of this, by means of peering relationships between pairs of clusters. This supports the composition of services across the clusters, as well as monitoring the network path in-between, for liveness as well as for performance metrics such as latency, bandwidth, etc. At the top level, we have composed services – service-level paths are formed as paths in the overlay graph. An example service-level path from a source to a destination, through services S1 and S2 is shown in the figure. Note that we also allow for "no-op" services in-between that simply provide connectivity, and do not add any application functionality.

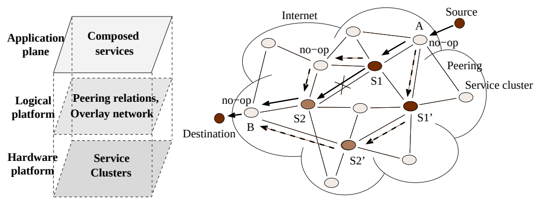


Figure 2. Architecture

The overlay network provides the context for exchange of performance information to create "optimal" service-level paths. Also, redundancy in the overlay network allows us to define alternate service-level paths for recovering from failures – the dotted lines between A and B in Fig. 2. The use of clusters amortizes the monitoring overhead across all client path sessions going through both peering service clusters. 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, as well as other requisite functionalities, are shown in Fig. 3.

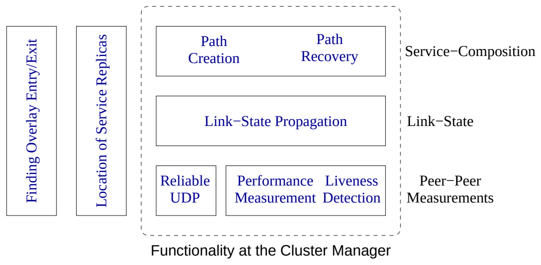


Figure 3. Software Architecture

A service-level path enters and exits our overlay network at entry/exit points (points A and B in Fig. 2). For a particular end-point outside the overlay network, the choice of the closest overlay node could be made using pre-configuration, or some simple selection mechanism. The first vertical layer in Fig. 3 captures this functionality. The next functionality we separate is that of service-location. This is the second vertical layer in Fig. 3. Here, we just need a list of locations of service replicas – something like the list of mirrors for a web-site. This can either be distributed slowly across the overlay nodes, or can be retrieved from a central (replicated) directory. The rest of the functionality in the figure resides in the CM of each overlay-node (service-cluster). The CM implements the mechanisms for *inter-cluster*, wide-area distributed service-level path creation and recovery. This is what we evaluate in this paper.

Functionalities at the Cluster Manager (CM)

The functionality at the manager node is in three layers (Fig. 3). The lowest layer implements communication between peer service-clusters in the overlay network, including performance measurement and liveness tracking. We have implemented liveness tracking as a simple periodic two-way heart-beat exchange, with a timeout to signal failure[2]. We have implemented latency as a performance measure – our architecture also allows measurement and exchange of other metrics such as cluster load, bandwidth, or other generic metrics.

At the next layer, global information about link performance and liveness is built using a link-state algorithm in the overlay network. We choose a link-state approach since we require global information for constrained path selection, the constraint being that the path should have a set of services on it, in a particular order.

The top layer implements the functionalities for service composition itself: initial creation, and recovery when overlay network failures are detected. We describe these algorithms along with our evaluations in Sec. 4. We note that service-level paths have an explicit session setup phase, and there is connection-state at the intermediate nodes. (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 means that, unlike Internet routing, failure information need not propagate to the entire network for corrective measures to be taken. We could have end-to-end path restorations or perform on-demand local-link recovery as in MPLS [8].

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.

Potential bottlenecks and sources of overhead

Each of the three layers presented above has overheads. Our main goal is to quantify these, identify sources of bottlenecks, and determine how quickly we can effect service-level path recovery. At the service-composition layer, while the presence of connection-state per path makes quick failure recovery easier, it could have scaling implications since a large number of client sessions may have to be restored on failure of an overlay link. Also, 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. The choice of a link-state algorithm is good for gathering global information. However, link-state algorithms consume more network bandwidth due to flooding, and this could be a potential source of bottleneck. At the lowest layer, failure detection itself is a concern when service-cluster peers monitoring one another are separated over the wide-area Internet. A conservative mechanism to detect failures could mean longer detection times in general, while a more aggressive mechanism may trigger spurious path restorations. We now turn to describing our evaluation testbed to study these overheads.

3 Experimental testbed

Evaluation of an Internet-scale system like ours is challenging since performance metrics such as time-to-recovery from failure

[2]We use a fixed timeout; we do not consider a more sophisticated adaptive timeout mechanism in this paper.

and scaling with the number of clients depend on Internet dynamics. A large-scale wide-area testbed is cumbersome to setup and maintain. Simulations are inappropriate since they do not capture processing bottlenecks. They also do not scale for large numbers of client sessions. We have developed a *network-emulation platform* for our experiments. We 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. This is done below the application. Such an emulation-based platform is provided by the Millennium cluster (www.millennium.berkeley.edu).

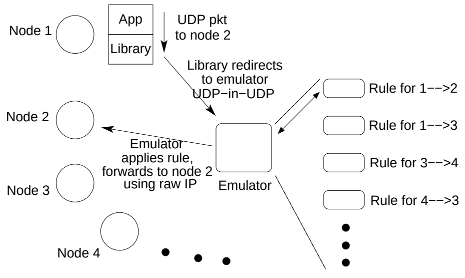


Figure 4. Emulator setup

For flexibility, we have implemented our own emulation packet modifier. The implementation is at user-level using raw-IP sockets and can handle UDP datagrams. In our setup, we 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 Fig. 4. 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. (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 Fig. 3). We have a real implementation of the algorithms – 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 (Fig. 4). In our architecture, the actual application data traffic does not pass through the CM. And hence in our emulation too, we only capture the control-traffic between the CMs. 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 Sec. 4.1.

Table 1 presents a brief characterization of our emulator setup. The emulator is setup on a Pentium-4 1500MHz machine with 256MB memory, and 256KB cache, running Linux 2.4.2-2. It is on a 100Mbps network. We have traffic passing through the emulator at a constant packet rate, with all packets being the same size. The emulator fires a randomly picked rule for each packet. In this setup, we measure the percentage of packets lost at the emulator.

In Table 1, the scaling limits of the emulator are reached in both dimensions – at large packet sizes and at high packet rates. We note that the emulator performs quite well for up to a packet rate of 20,000 pkts/sec, for pkt sizes below 500 bytes. This constitutes about $20,000 \times 500 \times 8 = 80 Mbps$, which is close to the ethernet

	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 1. % packets lost by the emulator

limit in our setup. 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 Evaluation

In this section, we turn to the evaluation of our system. In our set of experiments, we consider several metrics: (a) the time to *recovery* of client path sessions, after failure detection, (b) the time to *detection* of failures in Internet paths, (c) the additional control overhead due to spurious path restorations, and (d) other memory, CPU, and network overheads in our software architecture (Fig. 3). We study client session recovery time as a function of the number of client sessions (load) at each CM. We analyze two different recovery algorithms in Sec. 4.2 and Sec. 4.3. For these set of experiments, we use realistic modeling of Internet delay, but use controlled link failures. We then turn to a trace-based study of Internet path failure behavior in Sec. 4.4, and look at failure detection. Using this trace data, we study the time to path recovery under realistic Internet failure patterns in Sec. 4.5. This allows us to examine spurious path restorations. Finally, we look at other sources of overhead in our system in Sec. 4.6.

4.1 Parameter settings for the experiments

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.

The Overlay Network Topology

We use the following procedure to generate the overlay network. We first generate an underlying *physical network* with a Transit-Stub topology. This graph has a total of 6,510 nodes, with 14 transit ASes, each with 15 nodes, 10 stub-ASes per transit-node, and 3 nodes per stub-AS [19]. This topology is generated using the GT-ITM package. We then select a random subset of N nodes from this physical network to generate an N-node overlay topology. Next, we examine pairs of overlay nodes in the order of their closeness and decide to form peering relations between these. These peering relations correspond to overlay links in our architecture (Fig. 2). Overlay links are thus equivalent to *physical paths*. In this process of peering, 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 effect of Internet cross-traffic in our emulations. Each rule at the emulator involves these two parameters. Our mechanisms for path choice optimize the metric of path latency from entry to exit.

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

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

In our modeling of latency variation, we do not include occasional, isolated RTT spikes that do happen [2]. 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 Sec. 4.4 agree in spirit with observation (2) above – in our experiments, we observe that outage periods lasting beyond 1-2sec 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 postpone a discussion of this emulation to Sec. 4.4.

4.2 Time to path recovery: end-to-end recovery

The Algorithm

We first consider a flavor of path recovery that we term end-to-end recovery. Referring back to Fig. 2, consider the failure between services S1 and S2. In end-to-end recovery, two steps are involved. The downstream node (S2) detects the failure, and passes the information to the exit overlay node (B). The exit node then initiates a new path computation, which results in the path through S1' and S2' (dotted lines at the bottom of the figure). The path computation is a graph algorithm. The metric of distance in our implementation is edge-latency, which is additive. The well known algorithm for min-cost path computation over an additive metric is the Dijkstra's algorithm. We cannot apply this algorithm directly, since in our case the path has constraints that it has to pass through a given set of services. We have developed and implemented a generic $(k+1)$-stage modification to Dijkstra's algorithm – this gives the shortest "constrained" path, when we need k intermediate services.

This path computation is done by the exit node during path creation as well as end-to-end path recovery. (During path creation, the client sends its request for path creation to the exit node directly). After path computation, the exit node sends control messages "upstream" to setup the original or recovery path. The messaging for path creation and failure recovery for each client path session is done independently. Also, as mentioned in Sec. 2, all the computations and control messaging described above are done at the cluster manager of each overlay node. Hence 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.

Experiments

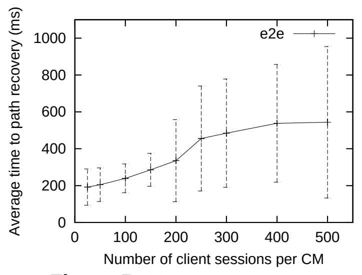
There are two components to the above algorithm: path computation, and the distributed messaging. Here, we study the latencies in the distributed messaging required to effect end-to-end recovery (path computation is evaluated in Sec. 4.6). In particular, we study the effect of scaling the system with respect to the number of client path sessions. We capture our metric of time-to-recovery of client sessions as a function of the number of client sessions that "terminate" at each overlay node – that is, the number of client sessions for which the overlay node is the exit node, and hence its CM is responsible for path creation and recovery for that session. Note that the number of client path sessions passing through (as opposed to terminating at) an overlay node will be higher since each client path passes through many overlay nodes. In the rest of the discussion, we refer to the number of client sessions terminating at each overlay node as the *load L* on it (or its CM).

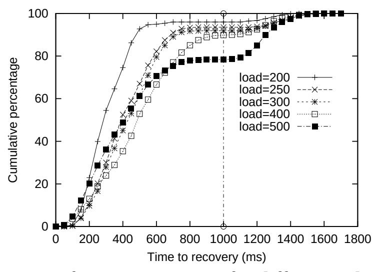
In this set of experiments, we use a 20-node overlay network generated as described earlier. This graph has 54 edges. There are a total of ten different services, each with two replicas in the overlay network. Each client path session involves two different services from among these. Note that although we have only two logical services, the path could stretch across many more overlay nodes, via the no-op services.

Across the runs, we vary the load *L* from 25 to 500 paths per CM, with equal load at all the 20 CMs. For a given load, we first establish all the paths (total #paths = #paths terminating at a CM × 20 CMs). The pair of services for each of the client paths is chosen at random. We then deterministically fail the link in the overlay network with the *maximum* number of client sessions traversing it – 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. 5(a) 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 600ms even for a load of up to 500 paths per cluster manager. Secondly, this average increases only slowly as the load increases – this suggests that the system has not reached its saturation point yet. Thirdly, the variance of the time-to-recovery across all failed paths is large at high load. To explain this, we plot another graph.

Fig. 5(b) shows the CDF of the time-to-recovery of the failed paths for different values of the load. We see that the majority of the paths recovery well within 1sec, and a small fraction of the paths take over 1sec 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





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

Figure 5. (a) Time to recovery vs. Load (b) CDF of time-to-recovery for different values of load (c) Detecting the bottleneck

of Fig. 3. This layer implements a re-transmit after 1sec, if there is no reply to the first packet[3]. 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. 5(a).

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. Also, the control packet losses could not be because of the deterministically failed link, since our algorithm does not send any recovery messages on the failed link itself). Case (1) would mean 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 100ms 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 1.

In Fig. 5(c), 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 network in-between) loses a small number of packets, and this number increases with the number of paths in the system. The table also gives the maximum rate seen by the emulator in 100ms windows. Referring back to Table 1, 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 1.0sec.

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 in Fig. 5) 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.

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. For comparison, the text-to-speech service we implemented in [15] 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 one final 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 links was highly skewed. The system can be expected to scale even better if a load balancing metric such as cluster-load is used. One of our immediate future plans is to work with such a metric.

4.3 Time to path recovery: local recovery

The Algorithm

We now examine an alternate method of recovery which we call “local” recovery. This is illustrated in Fig. 2 – the dotted lines between S1 and S2, above the original path. Here, the idea is that we patch a path locally by routing around the failure. The downstream node detects the failure, and finds an alternate path from its peer node, to replace the failed link. It then sends signaling messages along the new *local* alternate path, to be added as a patch to the original 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.

Experiments

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

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

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. 6(a). 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 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. 6 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. 5(a). 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, we note that the time-to-recovery has low values, around 700ms, as earlier. Also, the variance in the time-to-recovery goes up with load, as in Fig. 5(a). 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 200-300ms. (As our discussion in Sec. 4.4 will show, these small differences will be dwarfed by the time to failure *detection* in Internet paths – about 1.8sec).

The second graph 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 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. 4.1. 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.

4.4 Modeling Internet failure behavior

So far in our experiments, the failures in the overlay links have been artificially introduced. We have not realistically modeled how often Internet path failures happen, or how long they last. While this allowed us control over our experiments to understand the system behavior, we would like to see our system performance given realistic Internet path failure patterns. Further, an aspect we have not addressed so far is, how quickly failures can be detected, reliably. We turn to these issues now.

Failure detection: A key aspect of our system is its ability to detect failures in Internet paths. To achieve high-availability, we need to detect failures quickly. 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. An example is shown in Fig. 2 – the first leg after S1. This is important since unlike the telephone network, the Internet paths are known to have much lesser availability [13, 12].

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. There is a notion of a *false-positive* when the receiver concludes failure too soon, due to an intermittent loss. We term a path restoration triggered by such a false-positive to be a *spurious* path restoration. There is a trade-off between the time to failure detection and the rate of false-positives. If the timeout is too small, failures are detected rapidly, but the false-positives increase, and vice-versa when the timeout is too large. We study this in detail now, using wide-area trace data.

Trace data: For our purposes, we need a model for the incidence and duration of failures. There have been studies of failures or packet loss patterns at small time scales (less than 1sec) [18, 6]. These have shown that there is correlation of packet loss behavior within one second, but little correlation over a second. Further studies have estimated failures that last for over 30sec [20, 7]. 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 with a period of 300ms between pairs of geographically distributed hosts. 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. The heart-beat exchange was done for an extended period of time – for 3-7 days for the 9 pairs of hosts.

To understand the nature of Internet path outages, we compute the gaps between successive heart-beats at the receiving end. Given the heart-beat period of 300ms, we attribute gaps of length below 600ms to jitter in the arrival times. And, we attribute gaps of 600ms and above to outages in connectivity. Looking across all gap-lengths in an experiment, we get a distribution. Fig. 7(a) shows this distribution as a CDF for 3 pairs of hosts. Note that the y-axis starts from 99.9%. (The plots for other host-pairs are similar and we do not show them here).

We observe from the data that there is a sharp knee in the CDF between about 1.2-1.8sec for most graphs. This means that there are very few temporary outages that last longer than the knee-point. To see this more clearly, we view the data in another form. We compute the rate of occurrence of gaps of a given duration, averaged across an entire trace. Fig. 7(b) plots this rate of occurrence on a log scale, for various values of the outage duration on

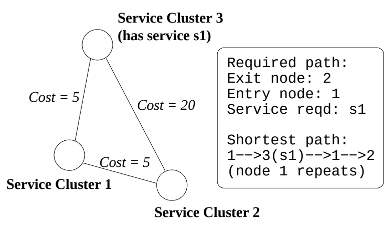
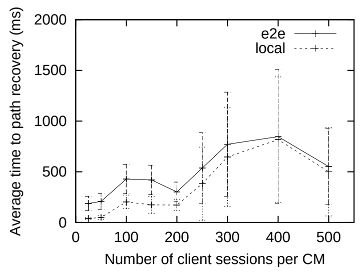
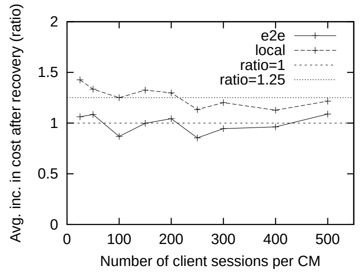


Figure 6. (a) Node repetition: an example



(b) Local vs. E2E recovery (time-to-recovery)



(c) Local vs. E2E recovery (path cost)

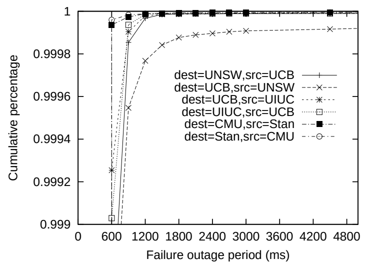
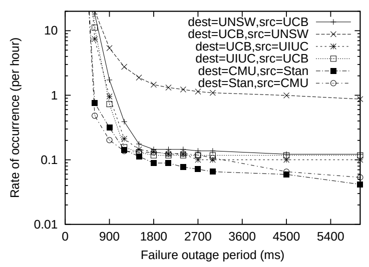
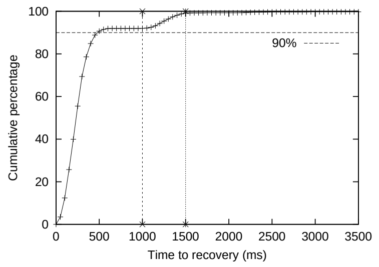


Figure 7. (a) Gap distribution (CDF)



(b) Outage occurrence rate



(c) Performance under realistic failures

the x-axis. Even in the log scale, there is a sharp knee around 1.2-1.8sec for all the plots. We use this set of data in two ways: (1) we use the distributions in Fig. 7(a) to model the distribution of outage periods on the Internet, and (2) we use the empirical value of 1.8sec, as suggested by the knee-points in either plot, as a timeout to conclude failures. We now return our discussion of path recovery time, but with the above modeling of Internet failures.

4.5 Performance under realistic Internet failure behavior

In this experiment, we wish to study two things: (a) the extent of spurious path restorations under realistic Internet outage patterns, and (b) the performance of our recovery messaging under realistic 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 the graph (Fig. 7(a)). For an overlay link in the testbed, we choose one of the 18 distributions at random. We have a fixed timeout of 1.8sec 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 15min.

During the run, across all the 54 edges in the graph, there are 162 outages that last 1sec or more, of which 32 outages last 1.8sec or more, and 7 last for 20sec or more. There are 11,079 end-to-end recovery attempts triggered. This represents an average of about 2 recoveries per path 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 3sec.

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. 4.2. 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: $\#outagesover1.8sec/\#links/15min = 32/(54 \times 2)/0.25 = 1.2/hour/link$. 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 path recovery.

Referring to the graph in Fig. 7(a), if we had a failure detection time close to 1sec or less, we are much more likely to have many spurious restorations. At 1sec on the x-axis, we are not yet yet beyond the knee-point. On the other hand, if we have a failure detection timeout much longer than 1.8sec (beyond the knee), the number of spurious path restorations might go down, but we will have a much longer time-to-detection of failures. Since we have a manageable frequency of spurious restorations with a timeout of 1.8sec, this is a reasonable timeout to have for failure detection.

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. However, at higher loads, the rate of path restorations, and the fraction of spurious path restorations, could be important factors in the stability of the system. We plan to take a closer look at this aspect in the near future, using a more scalable, distributed emulation platform.

Fig. 7(c) shows the CDF of the time-to-recovery of all the paths. Note the flat region in the CDF, as in Fig. 5(b). 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 1sec. 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 [12].

4.6 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 300ms in our implementation. And peer-peer latency measurements are done once every 2sec. The bandwidth consumed by these is miniscule.

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 60sec 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 15min, 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 [4].

Another possible source of overhead is the graph computation involved during path creation. In fact, the same graph computation is invoked during path recovery as well. 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. The complexity of our $(k+1)$ stage modification is $(k+1) \times E \times log(N)$. 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 50ms, and only about 3MB of memory. This figure of 50ms could be significant overhead if this computation is done for every path 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 subsection. Since we do not run this algorithm for every path creation/recovery, this is not a source of bottleneck.

4.7 Summary

In summary, our results show that failure recovery can be performed in our overlay network of service clusters, within 1sec for over 90% of client sessions (Sec. 4.5). Our trace-data, and the experiments using those show that failure detection can be quite aggressive, with a timeout as low as 1.8sec, 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 [12]. This figure of 2.8 seconds is definitely good enough for real-time, but non-interactive applications, which usually buffer about 5-10sec of data. For interactive applications, 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.

5 Related Work

The idea of service composition itself is not novel, a simple example being unix piping. The TACC project [10] developed models for fault-tolerance of composed services within a single cluster. The solution is based on monitoring using cluster-managers and front-ends. Apart from the TACC model, cluster-based solutions for fault-tolerance have been studied for other kinds of applications as well. The Active Services [3] model uses a soft-state mechanism for maintenance of long-lived sessions. The LARD approach [14] does load-balancing of client requests for a web-server within a cluster. However, such cluster-based approaches do not address performance or robustness across the wide-area.

In the context of web-servers, the problem of selecting an appropriate service instance in the wide-area based on network and server performance has been studied by earlier approaches [17, 9]. However, for composed services, we have multiple "legs" of the service-level path, and we need to optimize the overall composition, and not just one leg of it. Also, web-server selection mechanisms do not address fail-over for long-lived sessions, since web-sessions typically last for a short period of time (a few seconds).

Routing around failures (above the IP level) in the wide-area has been addressed in other contexts. Content-addressable networks [16] provide an overlay topology for locating and routing towards named objects. The RON project [4] also 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 [5]. 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.

The IETF OPES group [1] defines an architecture for "open services" that can be "plugged", or composed. However, this architecture does not include mechanisms for *recovery* when a composed session fails. ALAN [11] proposes application-layer routing by proxylets. The operational model there 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.

A unique aspect of our work is the use of an emulation-based testbed for evaluation. Most systems in the networking world are evaluated using either simulations or real experiments – neither of these approaches is suited for our purposes. Our emulation testbed using the Millennium cluster of machines has allowed better modeling than simulations, and more control than real experiments.

6 Conclusions and Ongoing Work

We started with the goal of being able to compose services in a robust fashion, providing recovery mechanisms for long-running client sessions. Our architecture for this is based on an overlay network of service clusters. In this paper, we have evaluated our architecture for its primary goal of quick recovery of client sessions. Our approach is based on a system distributed across the wide-area Internet. Its evaluation presents a challenge since a simple simulation-based approach would not only have been unrealistic, but would also have failed to identify the bottlenecks in a real system implementation. Developing and maintaining a large scale testbed across the wide-area Internet would have been cumbersome, and would not have been suited for a controlled design study. Our emulation-based approach has allowed a controlled design study with a real implementation. The control overhead in our software architecture is minimal, and requires little additional provisioning. Our trace-driven emulation shows that our recovery algorithms can react within 1sec for most (over 90%) of the client sessions. Network failure detection itself can be done with a couple of seconds, with a manageable frequency of spurious path restorations. Our ongoing work includes improving our emulation testbed to perform further scaling experiments, and to study issues of load-balancing and stability in path restoration. We have also developed a set of composable services [15]; we plan to test the usefulness of our recovery algorithms for these applications.

Achnowledgements: We thank L. Subramanian, S. Machiraju, A. Costello, S. Agarwal, and the anonymous reviewers for comments on earlier versions of this paper. We thank M. Baker, M. Roussopoulos, J. Mysore, R. Barnes, V. Pranesh, V. Krishnaswamy, H. Karl, Y-S. Chang, S. Ardon, and B. Thai for helping us with the wide-area trace collection.

References

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

[2] 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.

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

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

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

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

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

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

[9] S. G. Dykes, C. L. Jeffery, and K. A. Robbins. An Empirical Evaluation of Client-side Server Selection Algorithms. In *IEEE INFOCOM*, Mar 2000.

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

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

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

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

[14] 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.

[15] 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.

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

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

[18] 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.

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

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

Bhaskaran Raman received his B. Tech in Computer Science and Engineering from Indian Institute of Technology, Madras in May 1997. He received his M.S. in Computer Science from University of California, Berkeley, where he is currently a Ph.D. candidate. His research interests are in communication networks, large-scale Internet-based systems, and Internet middleware services.

Randy H. Katz received his undergraduate degree from Cornell University, and his M.S. and Ph.D. degrees from the University of California, Berkeley. He joined the faculty at Berkeley in 1983, where he is now the United Microelectronics Corporation Distinguished Professor in Electrical Engineering and Computer Science. He is a Fellow of the ACM and the IEEE, and a member of the National Academy of Engineering. His current research interests are Internet Services Architecture, Mobile Internet, and the technologies underlying the convergence of telecommunications and packet networks.