Load Balancing and Stability Issues in Algorithms for Service Composition

Bhaskaran Raman, Randy H. Katz

***Abstract*— Service composition enables flexible creation of new services by assembling independent service components. We are focused on the scenario where such composition takes place across the wide-area Internet. We envision independent providers deploying and managing service instances and portal providers composing them to quickly enable new applications in next-generation networks.**

One of the important goals in such service composition is load balancing across service instances. While load balancing has been studied extensively for web-server selection, the presence of composition presents new challenges. First, each client session involving composition requires a *set* of service instances and not just one server. Second, unlike web-mirror selection, we also concern ourselves with load balancing in the presence of failure recovery *during* a client session. We introduce (a) a metric to choose the set of service instances for composed client sessions: the *least-inverse-available-capacity (LIAC)* metric, as well as (b) a *piggybacking* mechanism to give quick feedback about server load. We then introduce an additional factor in the load balancing metric to avoid choosing far away service instances. Our experiments, based on an emulation testbed, show that our load balancing mechanism works well under a variety of scenarios including network path failures.

I. Introduction

Service composition is the process of assembling independent, reusable service components to construct a richer application functionality. In the context of next-generation networks, with services deployed and managed by independent providers, composition can enable rapid development of new applications. We term the set of service components in a particular composition, along with the network paths in-between as a *service-level path*. When service components are deployed by multiple providers, the service-level path could stretch across the wide-area Internet, across Internet domains. In the scenario of Internet-wide deployment, it is important to address issues of scale, load-balancing across service instances, and stability.

Fig. 1 captures the scenario under consideration. There are several replicas of different services at multiple Internet locations. A client session is formed by choosing a specific set of instances. Our focus is on the mechanisms for dictating this choice to achieve load balancing across the 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 [1], [2], [3], [4], [5], [6], [7] 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 paper is organized as follows. In the next section, we briefly present the background setting of our work and the problem statement. We then present our mechanisms for load balancing in Sec. III. We discuss the load balancing metric as well as the piggybacking mechanism for updating load information. Sec. IV presents experiments with our load balancing mechanism under a variety of scenarios. Sec. V discusses related work and Sec. VI presents concluding discussions.

II. Background

Service composition uses component services to enable new applications. The reusability of the independent components

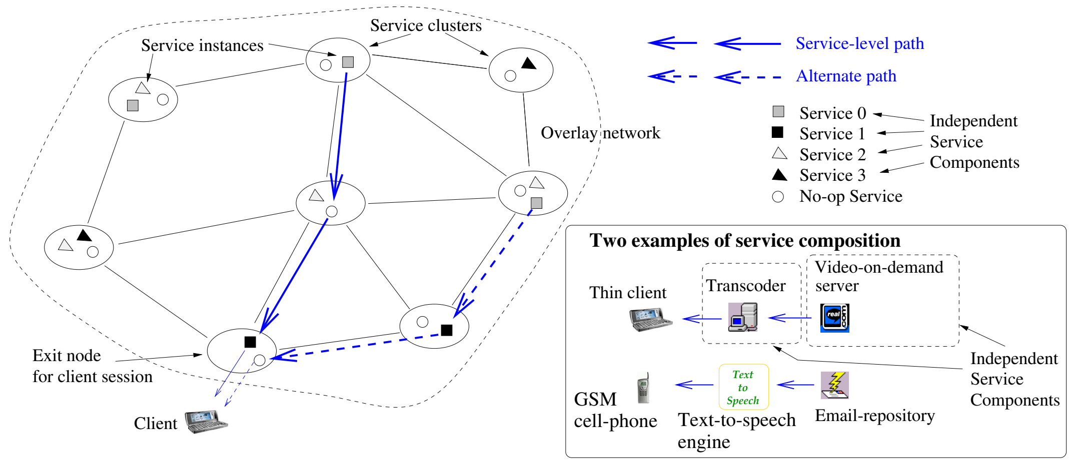

Fig. 1. Service composition: scenario under consideration

for different compositions gives flexibility. A simple example of the concept of composition, albeit in a different context, is that of combining independent programs using Unix piping. 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. Two specific examples of composition are shown in Fig. 1. The first is that of a composition involving a video-on-demand server and a video transcoder to enable video on a thin client. The second example composes a text source such as an email repository with a text-to-speech conversion engine to read out email to user on her cell-phone [8]. There are several other composed applications that we have built in previous work [9]; others are discussed in [10].

While there are several challenges in the context of service composition, in this paper we focus on load-balancing across the replicas of services placed at different Internet locations. In previous work [8], we have described an architecture for wide-area service composition. We now give a brief overview of the architecture to describe the context of the load balancing issue.

Our architecture is based on a service platform consisting of several service clusters deployed at different Internet locations (see Fig. 1). 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. 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 [8], we describe 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).

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 [11], [12] 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).

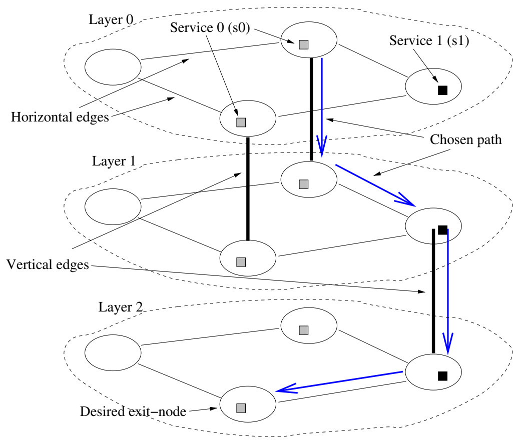

Fig. 2. Graph modification for service composition

In the context of programmable networks, a graph algorithm for constructing paths with intermediate processing sites in presented in [13]. This algorithm applies the well known Dijkstra's algorithm [14] for least-cost computation in a transformed graph. While [13] 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 [13], 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 [13]. 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. 2 where the client requests 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".

III. 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. III-A, 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. III-B, 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. III-C. We address this by introducing a "no-op" factor in the LIAC metric.

A. 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. 2), based on the information about the different nodes' load. This is 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 [15]:

$$PathCost = \Sigma_{link \,\epsilon\, path} \frac{1}{AvlbleBandwidth_{link}} \quad (1)$$

The intuition behind this metric is that the cost of using a particular link is inversely proportional to the bandwidth

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

available on it currently. That is, the closer to capacity a link is, the less likely that it will be used. The simulations in [15] 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:

$$PathCost \;=\; \Sigma_{S\ \epsilon\ path} \frac{1}{MaxLoad_S - CurrLoad_S} \quad (2)$$

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. 2); 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 1 favors network links with the maximum available bandwidth. We term the metric in Equation 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 have a real implementation of the algorithms and emulate wide-area latency. Our emulation platform is the Millennium cluster of workstations [16]. Each node in this 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 follows. We first generate an underlying physical network using the GT-ITM package [17], [18]. We then select a random subset of nodes to represent the location of the overlay network nodes. We then form the overlay network links by choosing the shortest paths along the original physical network. The latency on the overlay links is determined as follows. The GT-ITM package assigns costs on the edges of the graph it generates. We add up the edge costs along an overlay link to determine its cost. We then normalize this so that the maximum (one-way) latency along an overlay link is 100ms. This forms the base latency on an overlay link. We then vary the latency between this base value and twice the base value. The variation is based on results from a wide-area RTT-study [19]. The details are not important here since our results do not depend on this; the interested reader may refer to [8]. (Intuitively, load-balancing depends on the feedback loop, and not the finer variations in latency).

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 IV. 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 IV). 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 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 $2,500$[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 I 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

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

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. 3 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.

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 I

LOAD DISTRIBUTION ACROSS SERVER REPLICAS

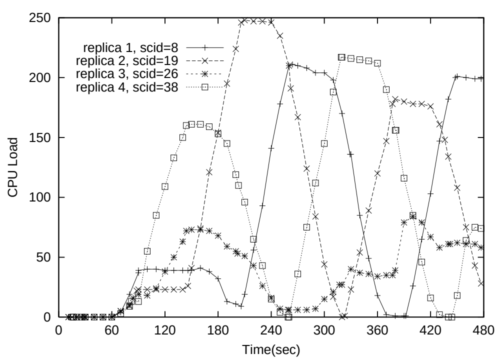

Fig. 3. Load variation with the load-balancing metric

We observe large variations over time in the load at each of the four replicas of the service. 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. 4. 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.

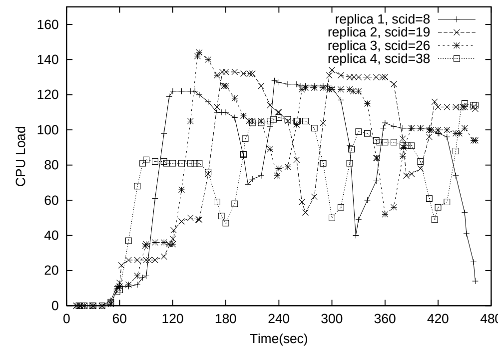

Fig. 4. Effect of lower link-state update period

The load variation also offers an explanation for the behavior observed in Table I. 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 I is replica #3 at node 26). This fourth replica thus get used much lesser than the other three.

B. Piggybacking

In the above experiments, we did not have the load go beyond a small fraction of $MaxLoad$. Such a setup allowed us to study the load variations without pushing the system to overload. We did run other experiments where we loaded the system close to its overall capacity, and many of the replicas experienced periodic overload due to the load variations. 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. The effect of such overload is application dependent. For some applications, the new requests may have to be rejected altogether due to overload, while for others, the overall performance goes down with overload. In either case, load variations and the resultant overload conditions are undesirable. We now look at how load balancing can be achieved.

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).

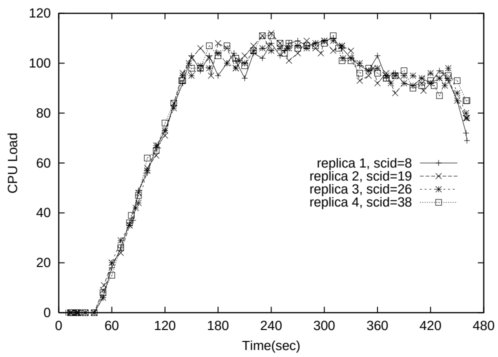

Fig. 5. Effect of piggybacking load information

We experiment the performance of this piggybacked load update mechanism with the same emulation setup as in the previous sub-section. Fig. 5 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. Note that periodic load information updates cannot be done away with for this reason – they are required to correct load overestimates.

C. *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. 6 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.

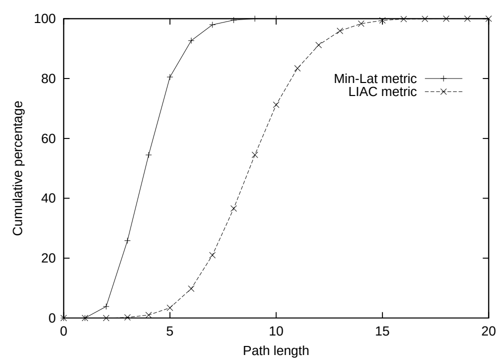

Fig. 6. CDF of path lengths: comparison

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. 2), 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:

$$PathCost = \Sigma_{S\ \epsilon\ path} \frac{1}{MaxLoad_S - CurrLoad_S} + \Sigma_{(D,U)\ \epsilon\ path} \frac{\alpha}{MaxLoad_D - CurrLoad_D} \quad (3)$$

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 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. IV-B).

Fig. 7 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).

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. 8 shows the load variation for this case with the use of the metric in Equation 3. We use a link-state update period of 60 seconds again. We see that the load variation

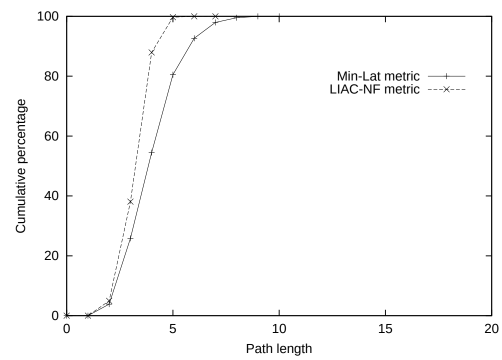

Fig. 7. Comparison of path length CDFs, with $\alpha = 0.1$

is still very less in comparison to Fig. 3, where we had no piggybacking. We observe more variations than in Fig. 5 – 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.

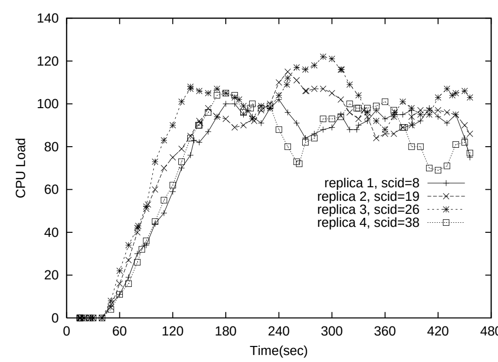

Fig. 8. Load variation with piggybacking, with no-op factor $\alpha = 0.1$

IV. 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, where a portion of the network is constantly loaded more than the rest of the network (Sec. IV-A), (b) effect of varying α as well as the number of service replicas in the network (Sec. IV-B), (c) effect of increasing the size of the network (Sec. IV-C), 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. IV-D & IV-E).

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.

A. *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. 9 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.

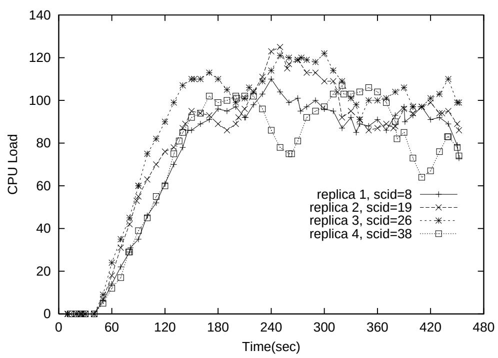

Fig. 9. Load variation with uneven incoming load

B. *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. 10 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.

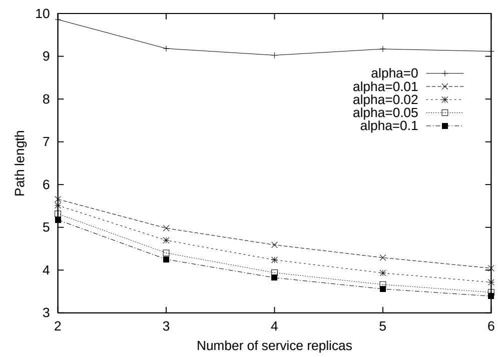

Fig. 10. Path length variation with α

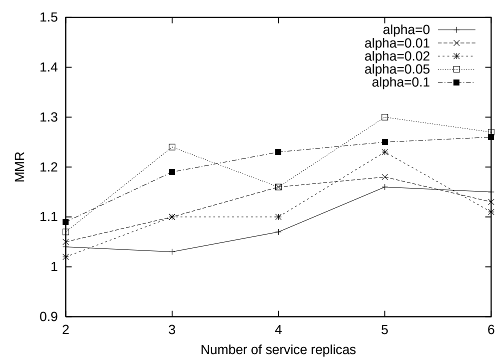

Fig. 11. Max-min-ratio (MMR) for different values of α

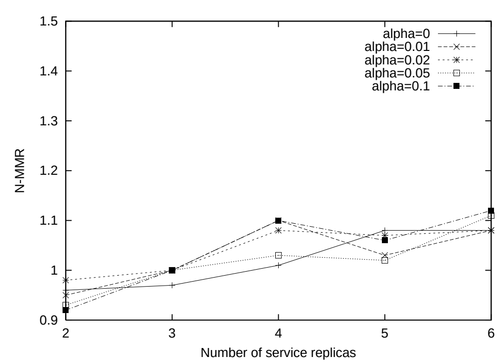

Fig. 12. Next-to-max-min-ratio (N-MMR) for different values of α

Fig. 11 shows the variation of MMR with the number of service replicas, for different values of α, and Fig. 12 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.

C. 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 subsection. Instead of showing these for a single service “s0”, we show it averaged across all the 20 kinds of services “s0”-“s19”. Fig. 13 shows the two ratios as a function of the number of overlay nodes. Fig. 14 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 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.

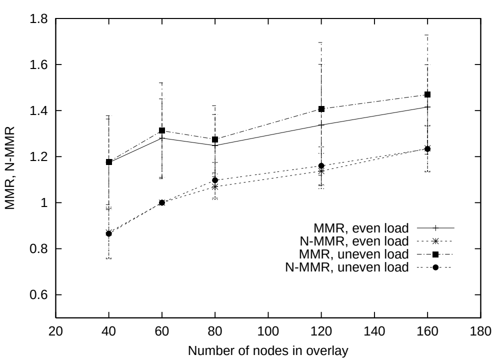

Fig. 13. MMR, N-MMR for different overlay sizes

D. Load balancing and failures

One of the primary goals of our architecture is the recovery of client path sessions on network failure. In [8], 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

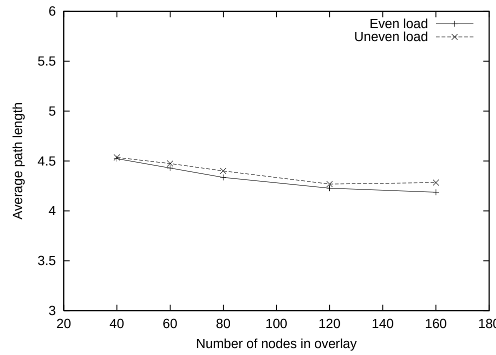

Fig. 14. Path length for different overlay sizes

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, 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.

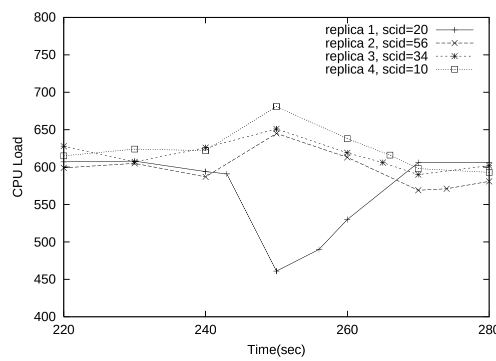

Fig. 15. Load variation under failure/recovery: s8

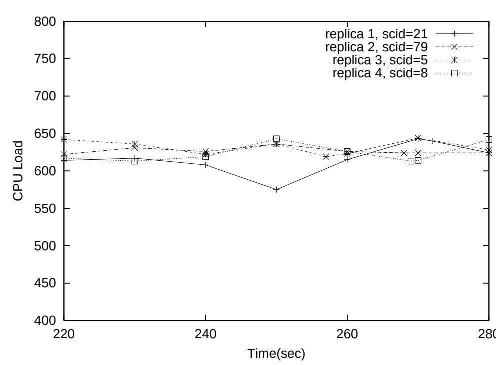

Fig. 16. Load variation under failure/recovery: s0

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. 15 shows the load variation for the service "s8" and Fig. 16 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 [8]. 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 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".

E. 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. 17 and Fig. 18 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.

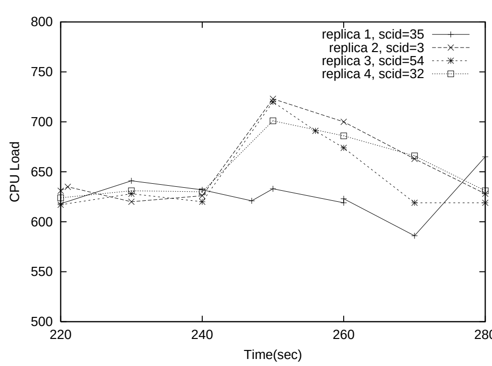

Fig. 17. Load variation under simultaneous failure/recovery: s5

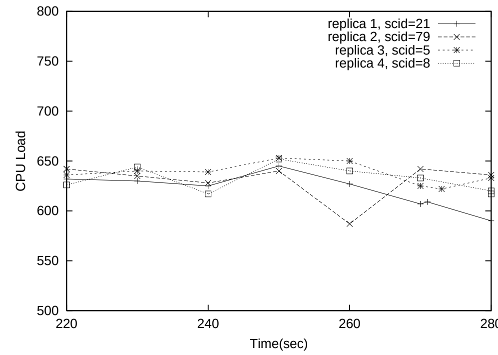

Fig. 18. Load variation under simultaneous failure/recovery: s0

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 case of the service with a replica close to the failure. The explanation for these remains the same too. One difference between Fig. 15 and Fig. 17 that can be observed is that the load at the replica that gets left behind remains flat for a longer time in Fig. 17 than in Fig. 15. 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. 15 and Fig. 17 is not due to the double-link failure in Fig. 17.

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. 15, node 20 happens to be an exit node for a larger fraction of the failed paths, whereas in Fig. 17, 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.

V. Related Work

The issue of load balancing among server replicas has been considered in past work. The mechanism for the intelligent redirection of client requests to machines within a cluster [12], as well as mechanisms for the choice among wide-area distributed replicas have been studied. Several mechanisms have been proposed for load balancing of distributed web-server systems [1]. These include client-based approaches [2], [3], [4], DNS-based approaches [5], [6], as well as dispatcher-based approaches [7]. 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. Our architecture, the use of metric-based choice of service-level paths, and the piggybacking mechanism distinguish our work from previous research on web-server selection.

Path selection in a network has been studied in QoS literature [15]: clients request bandwidth between two points in the network and the goal is to minimize call blocking rate. The least-distance metric is known to perform well in this context [15]. While the problem considered in [15] is quite different, we borrow intuition for the LIAC metric from this. Our use of the piggybacking mechanism is novel. Also, we have considered behavior under failure recovery, while this has not been considered in [15]. In the context of MPLS, failure recovery has been considered in path selection [20], but it uses pre-allocation and not dynamic path selection as in our case. Our work also differs from these in that we have considered "constrained" path selection – paths with intermediate services in them.

An algorithm for the construction of paths with intermediate services is presented in [13], albeit in a different context. We leverage this in our work, and address the important aspects of setting the costs for the transformed graph in the algorithm to reflect load at the service replicas. We also study the interaction with the load information propagation mechanism (link-state updates and the piggybacking mechanism).

VI. Summary and Conclusions

Service composition is a way to enable flexible creation of new services through the use of existing service components. In this paper, 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 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.

There are several other scenarios that we have not considered and deserve further exploration. First is the issue of scale of the overlay network. Our graph algorithm is based on the Dijkstra's algorithm and takes $O(E \times log(N)$ time, where E is the number of edges in the network, and N is the number of nodes. In practice, for a few thousand node graph, this could translate to about 50ms on a commodity PC [8]. To address this, one can think of maintaining an "active" set of close-by replicas for each service, and hence not consider the entire graph. This merits further study. Another issue relates to the behavior of the piggybacking mechanism in the presence of failure recovery of service-level paths. It would be interesting to explore the usefulness of artificially delaying the recovery of a fraction of the paths. The fraction of failed paths that recover immediately would provide piggybacked feedback of current load at the different replicas. This could avoid the temporary load variation that happens during failure recovery.

Acknowledgments: We thank Lakshminarayanan Subramanian, Kameswari Chebrolu, Adam Costello, Sridhar Machiraju, and the anonymous reviewers for their comments on earlier versions of this paper.

References

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

[2] C. Yoshikawa *et al.*, "Using Smart Clients to Build Scalable Services," in *Usenix*, Jan 1997.

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

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

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

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

[7] G. D. H. Hunt *et al.*, "Network Dispatcher: A Connection Router for Scalable Internet Services," *Computer Networks and ISDN Systems*, 1998.

[8] B. Raman and R. H. Katz, "Emulation-based Evaluation of an Architecture for Wide-Area Service Composition," in *SPECTS*, Jul 2002.

[9] 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.

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

[11] A. Fox, "A Framework for Separating Server Scalability and Availability from Internet Application Functionality," Ph.D. dissertation, U.C.Berkeley, 1998.

[12] V. S. Pai *et al.*, "Locality-Aware Request Distribution in Cluster-Based Network Servers," in *ASPLOS*, Oct 1998.

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

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

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

[16] Millennium, http://www.millennium.berkeley.edu/.

[17] "Modeling Topology of Large Internetworks," http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html.

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

[19] A. Acharya and J. Saltz, "A Study of Internet Round-Trip Delay," University of Maryland, College Park, Tech. Rep. CS-TR 3736, UMIACS-TR 96-97, 1996-97.

[20] M. S. Kodialam and T. V. Lakshman, "Dynamic Routing of Bandwidth Guaranteed Tunnels with Restoration," in *IEEE INFOCOM*, Mar 2000.