A Network Measurement Architecture for Adaptive Applications

Mark Stemm and Randy Katz
University of California, Berkeley
{stemm,randy}@cs.berkeley.edu

Srinivasan Seshan
IBM T.J. Watson Research Labs
srini@watson.ibm.com

Abstract—The quality of network connectivity between a pair of Internet hosts can vary greatly. Some hosts may communicate over high bandwidth, low latency, uncongested paths, while others communicate over much lower quality paths. Adaptive applications can cope with these differences in connectivity by choosing alternate representations of objects or streams or by downloading the objects from alternate locations. In order to effectively adapt, applications must discover the condition of the network before communicating with distant hosts. Unfortunately, the ability to predict or report the quality of connectivity is missing in today's suite of Internet services.

To address this limitation, we have developed SPAND (Shared Passive Network Performance Discovery), a system that facilitates the development of adaptive network applications. In each domain, applications make passive application-specific measurements of the network and store them in a local centralized repository of network performance information. Other applications may retrieve this information from the repository and use the shared experiences of all hosts in a domain to predict future performance. In this way, applications can make informed decisions about adaptation choices as they communicate with distant hosts.

In this paper, we describe and evaluate the SPAND architecture and implementation. We show how the architecture makes it easy to integrate new applications into our system and how the architecture has been used with specific types of data transport. Finally, we describe LookingGlass, a WWW mirror site selection tool that uses SPAND. LookingGlass meets the conflicting goals of collecting passive network performance measurements and maintaining good client response times. In addition, LookingGlass's server selection algorithms based on application-level measurements perform much better than techniques that rely on geographic location or routing metrics. More than 90% of the time, our technique allows clients to download mirrored web objects within 40% of the fastest possible download time.

I. Introduction

Network heterogeneity is a fact of life in today's Internet. Bandwidth between hosts can vary from kilobits to hundreds of megabits per second; packet loss probabilities range from less than 1% to over 50% of packets and round trip times vary from less than one millisecond to thousands of milliseconds. The decentralized nature of the Internet makes it very likely that the Internet will continue to be as heterogeneous in the future. This heterogeneity makes it difficult for applications to determine in advance what the network performance between a pair of Internet hosts will be. Because of the wide range of observed network characteristics, applications have been forced to become *adaptive*, making different decisions based on current network conditions. Examples of these decisions include the following:

- Client applications presented with a choice of servers that replicate the same service can choose the server that offers the highest-quality client $\longleftrightarrow$ server network path. A specific example of this is Harvest [1], a WWW caching system where a cache retrieves a Web object from any one of a number of "peer" yet provide a way to intelligently choose among them.
- Applications that have a choice of content fidelity can keep response times constant at the expense of quality. For example, WWW clients can use a transcoding proxy (e.g. Transend [2]) to change the quality of Web objects to match available bandwidth. A client receiving a multimedia stream can use a real-time transcoding service (e.g. Video Gateway [3]) to change the data rate of a multimedia stream depending on network characteristics.
- Applications can provide feedback to the user about the expected performance to a distant site. For example, Web browsers can insert informative icons next to hyperlinks indicating the object's expected transfer time. Clients using search engines could post-process the query results and re-score the documents based on the expected time to download a page.

Many existing applications provide the mechanisms for adaptation but use relatively simple policies. For example, Harvest relies mainly on round trip time to select a peer, choosing the first peer cache that returns a positive response. Transend and Video Gateway have no automated policy at all and rely on manual user intervention to choose the most appropriate object representation or data rate.

To improve their adaptation policies, many applications need to determine in advance the expected network performance between a pair of Internet hosts. Unfortunately, this capability is currently missing from the suite of Internet services. To address this limitation, we have developed a system called SPAND (Shared Passive Network Performance Discovery) that collects application-specific network performance information passively from a collection of hosts, caches it for some time, and uses this information to estimate network conditions. By sharing this information with clients, SPAND enables applications to be more adaptive.

The use of shared, passive measurements creates many challenges for network performance prediction system as well as its clients. We have developed an HTTP server selection tool, called *LookingGlass*, that uses SPAND to help choose among a number of Web servers that mirror the same content. This application must balance the goals of maintaining an up-to-date repository of network performance information and providing good client performance. Despite these challenges, our measurement techniques and server selection algorithms perform much better than alternate techniques that rely on geographic location or routing metrics. More than 90% of the time, our tech-

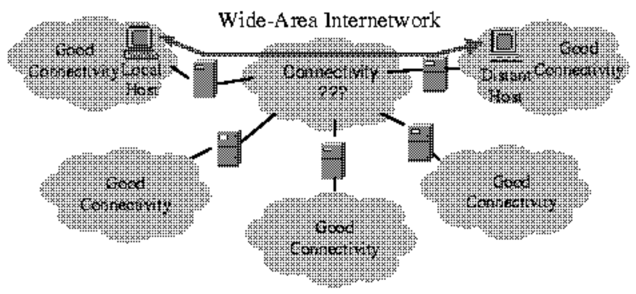

Fig. 1. Network Model behind SPAND. Hosts in well connected domains communicate with distant hosts through a network with unknown properties.

In this paper, we describe the SPAND architecture, how it works, where it works well, and how it has been used to facilitate the creation of adaptive networked applications. Section II describes the design choices and assumptions of SPAND and the types of applications that best benefit from using SPAND. Section III presents the SPAND architecture, focusing on how its extensible nature facilitates the integration of new applications. Section IV shows how the architecture has been applied to two different types of data transport applications: a generic bulk transfer application and an HTTP-specific application. Section V describes the LookingGlass application and its performance. Finally, we present our conclusions in Section VI.

II. Design Choices and Methodology

In this section, we discuss the network model and assumptions used in the design of SPAND. We also compare SPAND with alternate designs and discuss which applications are likely to benefit most from the SPAND architecture.

A. *Network Model*

The network model and abstractions that underlie our work are summarized in Figure 1. The network is abstracted into "domains" of high-bandwidth, low-latency, and relatively uncongested connectivity, each connected by a wide-area internetwork. The properties of network paths between domains are unknown, but are usually an order of magnitude lower quality than intra-domain paths. In addition, the quality of different wide-area paths can be very different.

In the next section, we show how this network model leads to the design of SPAND. Details about the design of an earlier version of SPAND as well as client access patterns are available in [4].

B. *Design Choices in SPAND*

Our architecture incorporates four important design choices that offer significant advantages over alternate approaches:

• First, our measurements are *shared*. Hosts explicitly share the measurements they make by placing them in a centralized per-domain repository.

The decision to use shared measurements follows directly from the network model. If two hosts in a local domain have high-quality connectivity to each other and lower-quality connectivity to some distant host, they can share performance in-

• Second, our measurements are *passive*. Instead of actively probing the network using ICMP packets or simulated connections, we rely on the traffic that applications generate as they communicate with other Internet hosts.

A significant advantage of passive measurements is that they do not introduce additional traffic into the network. Many active probing systems, such as Cprobes [5], Packet Bunch Mode [6], and Pathchar [7], use tens of kilobytes of probe traffic to generate a single performance measurement. This probe traffic is not directly used by any application and could be a significant portion of the traffic between a pair of hosts. Widespread use of active probing could significantly reduce the bandwidth available to useful application traffic.

An additional advantage of passive monitoring is that it measures performance for most popular destination hosts. Destination hosts that interact often with local hosts generate more traffic, and as a result, more network measurements.

• Third, our measurements are *application specific*. Instead of using network-level statistics such as routing metrics, latencies, or link bandwidths as the basic report in our system, we rely on application-level measurements such as response time or Web page download time to drive application decisions.

By measuring application-level performance, we are guaranteed to measure what the application is most interested in – end-to-end performance. Many existing systems use metrics such as routing information [8] [9], round trip estimates [1], geographic locality [10], and bandwidth [5] to determine the relative performance of different hosts. Some existing frameworks are more general but current implementations rely on routing metrics [11] or round trip time [12]. Unfortunately, these metrics often do not correlate with application-level performance. For example, a host that is many network hops away may perform better than a host that is closer. In fact, recent measurements from [13] have shown that some servers with high loads provide better application response times than less busy servers.

Other systems [14] [15] make application-level measurements of transfers initiated by "robots". However, these systems must ensure that the robot's transfers accurately reflect the usage patterns of actual clients. The robots may use transfers that are too long or too short, download a different set of Web pages than typical users, or space transfers so that they do not reflect an actual user's "think time".

• Lastly, our system is *extensible*. Our architecture makes it very easy to integrate new applications and new metrics into SPAND.

We cannot predict in advance all possible applications that may wish to use SPAND to measure and store performance information. Rather than providing a limited set of network performance metrics such as bandwidth, latency, packet loss probability, etc., we present a more database-like interface to applications and allow them to specify their own performance metrics. We continue to measure and store basic performance metrics

C. *Measurement Noise and How it Affects Application-level Decisions*

Although the shared, passive design of SPAND provides significant advantages, it also introduces significant challenges. The basic question that arises from these challenges is whether a shared, passive system can provide *accurate* performance predictions. All of the statistics measured in SPAND have a certain amount *noise* associated with them. This noise can be divided into several categories:

- *Network noise* is inherent in the state of the network. If a single client made back-to-back measurements of the state of the network, it would see some degree of variability in the statistics they measured.
- *Sharing noise* results from inappropriate sharing of network measurements between hosts and applications. By sharing information across hosts in a domain, SPAND assumes that these hosts observe similar application performance. However, this assumption may be incorrect due to differences in local clients' connectivity, differences in network stack implementations, or other differences between clients.
- *Temporal noise* results from the use of out-of-date network measurements that no longer reflect the true state of the network. Since SPAND relies on passive measurements, it has no control over the timing or frequency of network measurements. Unfortunately, measurements taken at a particular moment may not reflect the state of the network at some later time.

SPAND's use of shared, passive measurements assumes that these sources of noise are minor factors. To verify these assumptions, we performed some experiments designed to measure network, sharing, and temporal noise. Although not comprehensive, these experiments attempt to give a rough indication of how much noise is likely to come from network, sharing, and temporal sources.

C.1 Contribution of Network and Sharing Noise

To give a rough indication of the amount of network noise that can occur, we analyzed a packet trace collected at the gateway between the UC Berkeley Electrical Engineering and Computer Science Department and the rest of the Internet. This packet trace consisted of approximately 700,000 TCP connections between TCP senders in the Internet and TCP receivers inside UC Berkeley. The trace was collected between 6am on June 14, 1999 and 4pm on June 14, 1999. The trace consisted of 1516 unique UCB hosts and 22,710 unique distant hosts.

When post-processing the trace, we only considered TCP connections that transferred at least 1 kilobyte and only considered (sender,receiver) pairs that had at least 30 connections between them. We measured the duration of each TCP connection as well as the number of bytes transferred in the connection and divided these two measurements to obtain a generic "available bandwidth" network performance metric for each connection. To measure the variation in performance, we measured the fraction of transfers t where the performance for a given (sender,receiver) pair was more than a factor F away from the median performance for that pair. A value of t close to 1 indicates a great deal of variation and a value close to 0 indicates

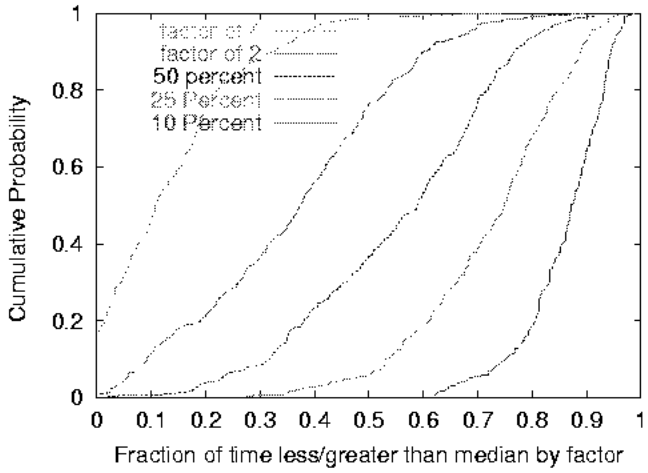

Fig. 2. Quantifying network noise. Figure shows likelihood of being more F (from left to right: 4, 2, 1.5, 1.25, 1.1) away from median performance for a given (sender,receiver) pair.

that may potentially have different available bandwidth measurements.

To quantify network, sharing, and temporal noise, we performed the following analyses with the trace. We first measured the amount of network noise in the trace by examining the variation in performance for individual (sender,receiver) pairs over a relatively small period of 30 minutes from 1pm to 1:30pm. We then assumed that the receivers would share performance information and again measured the variation in performance, this time for (sender,receiver) pairs over the 30 minute period. (The size of sharing group varied from a single client for unpopular distant hosts to hundreds of clients for more popular distant hosts). The difference in variation between these two cases allowed us to measure the sharing noise introduced by sharing performance information between receivers. We then repeated the experiment again by considering longer and longer time scales starting at 6am and continuing until 4pm. Examining the variation for longer time scales allowed us to measure the amount of temporal noise introduced by using potentially stale past performance information to indicate current performance.

Figure 2 shows the results of the first analysis. The x axis shows the fraction of time that the actual performance for a given (sender,receiver) pair was more a factor F away from the median performance observed for that pair of hosts, and the y axis shows cumulative probability. There are 5 curves on the graph, one for factors of (from left to right): 4, 2, 1.5, 1.25, and 1.1, respectively. We see from the figure that the available bandwidth for individual connections is almost always more than 10% away from the median bandwidth for a (sender,receiver) pair, and that differences of as much as 400% are common. This means that network noise limits the granularity of application-level adaptations to relatively coarse-grained decisions that differentiate between order of magnitude changes in performance.

Figure 3 shows the results of the second analysis. In addition to the curves in figure 2, the figure shows the degree of variation when local receivers share performance information and

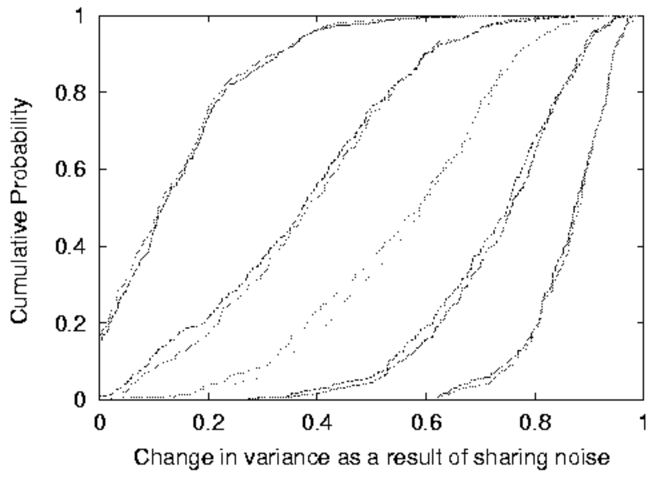

Fig. 3. Quantifying sharing noise. Figure shows likelihood of being more than F (from left to right: 4, 2, 1.5, 1.25, 1.1) away from median performance for a given (sender,group of receivers) pair.

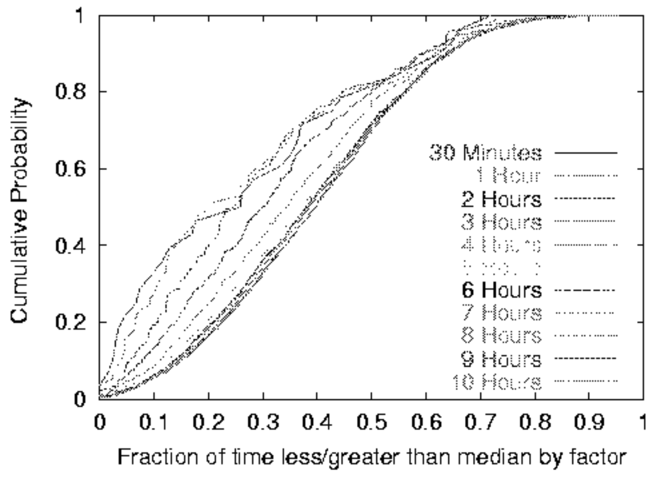

Fig. 4. Quantifying temporal noise. Figure shows likelihood of being more than a factor of 2 away from median performance for a given (sender,group of receivers) pair for increasing (from left to right) time scales.

left to right is the same as in figure 2). We see that for each factor F, there is only a small increase in variation from sharing performance information between receivers. This means that sharing noise is relatively unimportant when compared to network noise, and the benefits of sharing (increased availability of network performance information) outweigh the additional variation in performance.

Figure 4 shows the effects of different time scales on performance variability. For this graph, the factor F is set to 2 and the time scale varies (from left to right) from 30 minutes to 10 hours. The performance variability increases as the time scale grows to 6 hours. Further increases of the time scale beyond this do not affect the variability. For small time scales, however, the difference from the initial 30 minute period is modest. This implies that temporal variations can be significant, but over small

D. Applications Requirements

The SPAND architecture is designed to be extensible in order to support a wide variety of applications. In general, SPAND works best for applications that meet the following criteria:
- Applications that do not initially have enough network performance information to make a good adaptation decision
- Applications that cannot easily change their adaptation decision once it is made.
- Applications that do not have a way of sharing performance information with other instances of the same application

These criteria are more easily explained using some example applications. The following three examples meet the above criteria to various degrees:
- An adaptive WWW browser, where a client selects the version of a Web page best suited to it current performance.
- Bulk transfer applications, where a client selects one of a number of mirror sites to retrieve a single large object
- Streaming multimedia applications that use RLM [16], where clients select the number of layers they wish to receive from a multicast source

The WWW browsing application meets all of the above criteria. A client typically contacts one among large number of Web servers and downloads a few pages from it. This makes it unlikely that a client has enough information to select an appropriate representation to download. In addition, the client gains little by choosing an alternate representation for subsequent downloads if the original choice was inappropriate.

The bulk transfer application meets some, but not all, of the criteria. It is difficult for a single client to determine which mirror will result in the best performance. The client would either have to download the content from all of them simultaneously or select one at random. The client might be able to efficiently change its selection if the application supports mechanisms for retrieving sub-ranges of documents.

For both of the above applications, it is difficult to share performance information in a distributed manner, as the client applications may be short-lived and cannot participate in a distributed sharing algorithm.

The streaming multimedia application meets only few of the criteria and is not well suited to the SPAND framework. Although a client may not have much information about the initial number of layers to subscribe to, it can easily re-evaluate this decision based on network conditions, adding or dropping layers as necessary. Because executions of this application are relatively long-lived, it is also easy for the application clients to share performance information in a distributed manner. For example, the application could periodically advertise network characteristics using a separate multicast group.

III. The SPAND Architecture

In this section, we describe the components of SPAND architecture and how they communicate with each other. We also describe how the architecture is *extensible*, facilitating the support of newly developed applications.

A. Components of SPAND

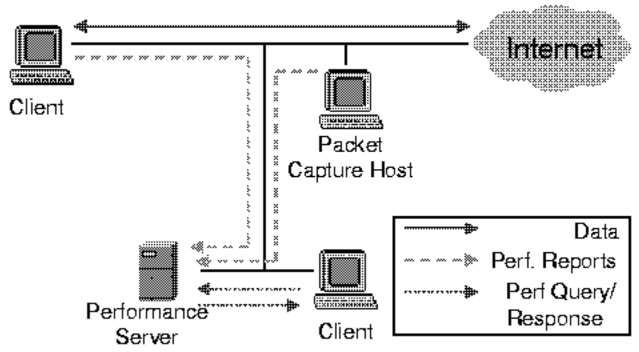

Fig. 5. Components of SPAND.

Servers, and *Packet Capture Hosts*.

A.1 Client Applications

Client Applications measure the state of the network and create *Performance Reports* that summarize the observed performance to distant hosts. They format of the reports are defined by each application to contain the information it is most interested in. For example, instead of reporting available bandwidth or round trip time, a Web client may report the amount of time taken to retrieve a particular Web object.

A.2 Packet Capture Host

In practice, it may be difficult to immediately upgrade all client applications to generate performance reports. To quickly capture performance from a large number of end clients, we employ a Packet Capture Host that passively observes all transfers to and from a group of clients (currently using BPF [17]). The Packet Capture Host determines an application's performance from its observations and sends reports to the Performance Server on behalf of the clients. The weakness of this approach is that the Packet Capture Host must use a number of heuristics to recreate application-level information that is available at the end client. Sections IV-A.2 and IV-B.2 describe some of these heuristics.

A.3 Performance Server

After creating Performance Reports, the Client Applications and Packet Capture Hosts send them to a Performance Server that stores them in a database. Client Applications also send *Performance Queries* to query a Performance Server for the observed performance to a distant host. The Performance Server responds with a *Performance Response* that indicates the typical performance seen by clients.

B. Messages Between SPAND Components

All messages between the components of our system use a format similar to Active Messages [18]. A SPAND message contains a handler string, an active flag, a data length, and a message-specific payload. If the active flag is 1, the handler string indicates the name of the function that should be used to and Performance Queries from Client Applications and Packet Capture Hosts are typically active messages, whereas Performance Responses from the Performance Server are typically not active.

To incorporate a new application into SPAND, an application writer implements active message handlers and integrates them into the Performance Server. The Performance Server is currently written in Java, so this process simply means placing the Java class files at a location where the Java Virtual Machine's class loader can find them. Although SPAND is extensible, the system does not allow arbitrary clients to upload active message handlers in real time as the Active Networks system [19].

Although applications can create arbitrary handlers for SPAND messages, most applications will provide at least the following basic handlers:

- *AddPerf*: Add the following Performance Report to the database. The data portion of the message contains the Report.
- *GetPerf*: Process the following Performance Request. The data portion of the message contains the Performance Request.
- *GetSumm*: Return a summary of the database for this application type. Usually, this includes the number of reports collected for each key (e.g. address, URL, etc.) in the database.
- *GetRaw*: Return the raw Performance Reports for the specified key (e.g. address, URL, etc.). The data portion of the message contains the key.

These basic handlers provide clients the ability to add and process reports.

IV. Realizations of the SPAND Architecture

In this section, we describe two realizations of the SPAND architecture. The first realization measures the performance of generic bulk transfers that use TCP as their data transport. The second realization measures performance for an HTTP-specific data transport.

A. Bulk Transfer Application

The goal of the bulk transfer realization is to measure performance for an application that uses a single TCP connection to send data from one host to another. We assume that the transmission of data is limited only by the speed of the network and not the speed of the sender or receiver of data. Our realization obtains the following metrics from the data transfer: the *available bandwidth* for the connection, the *round trip time* for the connection, and the *time to completion* for a specific transfer size of B bytes.

Although this realization does not measure true end-to-end application level performance, the statistics it measures are useful to large number of applications that use TCP connections for bulk data transport. The resulting system provides information about the current state of network communications and is similar to many other networking monitoring systems.

A.1 Bulk Transfer Metrics

The *available bandwidth* metric estimates the long-term bandwidth that a one-way transfer using a single TCP connection will receive, similar to the IETF IPPM Working Group's

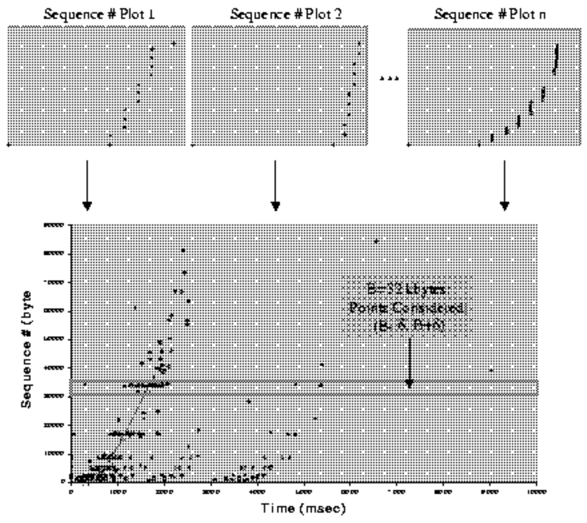

Fig. 6. Process to compute time-to-completion metric.

the transfer, including the initial SYN exchange but not including the FIN exchange.

The *round trip time* metric reports the smoothed round trip time between hosts in a local domain and a specific host in a distant domain. Individual round trip time samples are measured from the time when a TCP packet is sent out to when the acknowledgement for that packet is received. The individual samples are then exponentially smoothed using an algorithm similar to the one used by TCP [21].

The third metric, *time to completion*, is useful to applications that know a connection's transfer size in advance. We define time to completion as the time T needed to transfer B bytes using a single TCP connection, including connection set-up time. A significant advantage of this metric is that it is *non-parametric* and does not rely on an analytic model for the way a TCP connection should behave ([22] [23]). Since the metric reports observed connection dynamics, we do not need to worry if the models accurately predict TCP behavior.

This process to determine time to completion is shown in Figure 6. We start by combining sequence number plots for a large number of connections, resulting in a single scatter plot of sequence number vs. time. To aid in legibility, Figure 6 only shows those points where the number of bytes transferred had just surpassed a power of 2 (512, 1024, 2048, ...). On this scatter plot, we consider all points slightly above and below target number of bytes B, $(B-\delta, B+\delta)$. δ is dynamically chosen as the smallest δ for which at least 20 points fall between $(B-\delta, B+\delta)$. If δ is greater than 5 kilobytes, we report the statistic as not measurable. If δ is smaller than 5 kilobytes, then we take the median time value for all points within the region as the result T. The plot of time to completion in 6 show that the metric does a good job of following the "center of mass" for this set of connections. We also see that in the 64 Kbyte case, our

A.2 Obtaining Metrics

Because the Packet Capture Host is not located at end clients, it must infer how a TCP connection is used by an application. This can lead to inaccurate measurements of network characteristics such as bandwidth. For example, a Web browser may use a persistent or keep-alive connection to make many HTTP requests over a single TCP connection. Simply measuring the observed bandwidth over the entire TCP connection will include the gaps between HTTP requests in the total time of the connection, leading to a reduction in reported bandwidth. To account for this effect, the Packet Capture Host uses heuristics to detect these idle periods in connections. When the Packet Capture Host detects an idle period, it makes two reports: one for the part of the connection before the idle period, and another for the part of the connection after the idle period.

The Packet Capture Host also uses heuristics to make round trip time measurements. Because the Packet Capture Host lies between the endpoints of the TCP connection, it cannot directly measure the round trip time of the connection. Instead, it makes separate measurement of the round trip time from the local client to the Packet Capture Host (using data sent to the client) and from the Packet Capture Host to the distant host (using data sent to the distant host). The measurements for each component of the path are individually smoothed and then added together to result in a single SRTT measurement.

B. *HTTP Applications*

The goal of the HTTP realization is to measure performance for applications that use HTTP (on top of TCP) for data transport. Web browsers use HTTP to download Web pages that usually consist of an HTML document and several inline images. These objects are usually retrieved using multiple TCP connections. As a result, metrics that only report the performance of individual TCP connections are not as useful to applications as metrics that report full-page download times.

The most important reason to extend SPAND to have HTTP specific metrics is that the retrieval time for many Web pages is not limited by the network but by server performance. For example, many Web pages are produced as the output of server-side execution of Common Gateway Interface (CGI) programs. In addition, the performance of SSL transfers from secure Web sites is often limited by computation speed. Systems that only report network statistics would not be able to predict the difference in performance for these pages or servers. The ability to extend SPAND to incorporate server-side execution time is essential to building adaptive Web applications.

B.1 HTTP Metrics

To overcome these limitations, we measure and report *Web object download time* as the primary metric for HTTP applications. We define the Web object download as starting when the application initiates the transfer and ending when the application receives the last byte of content for the Web object. This also includes any server execution time as the result of CGI programs. In addition, the system records the download time for a reports together, SPAND reports the median download time as "typical".

B.2 Obtaining Metrics

The Packet Capture Host measures the Web download times by examining the HTTP headers and responses in TCP connections. This can be difficult: a single TCP connection may be used for a single HTTP transaction (for HTTP/1.0) or multiple HTTP transactions (for HTTP/1.1). In addition, the Packet Capture Host needs to link the multiple HTTP transactions that comprise a Web page to report its total download time.

The Packet Capture Host first recreates the ordered byte stream from individual TCP packets and parses the HTTP headers in the byte stream. It uses the `Content-Length:` field in HTTP Responses to identify the transfer of individual Web objects. The Web object download time is then measured as time between the SYN exchange of the connection and arrival of the last byte of the object. For subsequent HTTP requests using the same connection, we start the measurement with the transmission of the first byte of the HTTP request instead of the beginning of the SYN exchange.

In order to link together the transfers that comprise a Web page download, the Packet Capture Host uses the `Referrer:` field in each HTTP request. The `Referrer:` indicates the object from which the current request was referred. Therefore, the header for the transfer of each inline image contains the name of its associated HTML page. Although inclusion of this field is not mandatory, only a few Web clients such as JDK/1.1 and Pointcast do not include this header field. The Packet Capture Host uses a two-pass approach to recreate the Web page transfers. The first pass links together HTML pages and inline objects using the `Referrer:` field, and the second pass makes Performance Reports using the information gained from the first pass. We wait long enough between the two passes to assure that most Web pages have been completely transferred by the time the second pass starts.

C. SPAND Clients

We have developed several applications that use the bulk transfer and HTTP specific metrics described in the previous section. These applications were all developed using Muffin [24], an extensible HTTP proxy that allows the addition of customized filters for filtering HTTP request, response, and content streams. We briefly describe a few applications here.

We have added an HTTP request filter to Muffin that uses SPAND and HTTP Content Negotiation to reduce Web object download times [25]. The Content Negotiation protocol presents a Web client with a list of different versions of a single Web object and allows the client to select one of the variants to download. Our system uses transcoding to produce versions of the same Web object that are of different sizes and content fidelity. The request filter attempts to keep Web page download time fixed by selecting the best quality version that can be transferred in the target time. We have also added a content filter to Muffin that inserts informative icons in HTML documents that indicate the expected download time for hyper-linked objects (e.g. hyperlink. In addition, we have written several diagnostic tools that allow users to browse the contents of the Performance Server's database. Finally, we have written an HTTP mirror server selection tool called LookingGlass. The details of this tool are described in the next section.

V. LookingGlass: A Server Selection Application

One of the common methods to improve Web performance has been to replicate popular data items on multiple mirror sites. For example, the most popular download from www.download.com on Feb 1, 1999 (a chat program called ICQ) is available from 17 distinct sites in 13 countries. Similarly, the mirror page for www.redhat.com (http://www.redhat.com/mirrors.html) lists 82 distinct sites in 28 countries. There are several problems that arise from mirroring Web content at multiple locations:

- *Mechanisms for Mirror Advertisement:* There must be a way for Web servers to advertise where the mirrors are located.
- *Metrics for Mirror Ranking:* There must be a way to rank the mirror locations in terms of their quality of connectivity to local Web clients.
- *Mirror Selection Algorithm:* Given a ranking, there must be a way for Web clients to select the most appropriate mirror from which to download the object.

Currently, these problems are handled in manual and ad-hoc ways. To address the problem of mirror advertisements, the user is usually presented with an HTML page containing information about the mirror servers as well as links to the mirrored data. The administrator of the primary site must update this page every time a mirror is added or deleted. The mirror ranking process is performed by the end-user using metrics such as geographic location or advice from the Web site administrator. Unfortunately, such metrics do not necessarily indicate the best server for a user. The mirror selection algorithm is handled by the user clicking on a link on the HTML page of the mirror sites. This technique may lead to "hotspots" where all users attempt to download the object from the same mirror location.

A. The Solution: LookingGlass

LookingGlass is an HTTP server selection tool that addresses the above problems in the mirroring of Web content. It addresses the problem of mirror advertisements by automatically collecting information about the location of mirrored objects. It uses the Web object download metric for mirror ranking. However, instead of always selecting the server with the best metric, LookingGlass randomizes the selection with weightings in proportion to the metrics returned by SPAND. LookingGlass uses SPAND as the repository for both mirror location and mirror location performance information. In the following sections, we describe these mechanisms and algorithms in more detail.

A.1 Mechanisms for Mirror Advertisements

There are two major flaws with the current method for advertising mirrored objects. First, the mirrors are manually advertised by creating an HTML page. Second, the mirrors are transparent way to advertise mirrored objects, and a distributed algorithm for disseminating this information.

We can make the process of advertising mirrors transparent by using HTTP's Transparent Content Negotiation mechanism [26]. LookingGlass advertises alternate locations of a Web object by using the Content Negotiation "Multiple Choices" response to HTTP requests. The following example shows how this response describes the alternates and their characteristics:

```
HTTP 300 Multiple Choices:
Date: Tue, 11 Jan 1996 20:02:21 GMT
TCN: List
Alternates:
 {''http://us.imdb.org/index.html''
           {type text/html} {length 2176}},
 {''http://uk.imdb.org/index.html''
           {type text/html} {length 2176}},
 {''http://jp.imdb.org/index.html''
           {type text/html} {length 2176}}
```

When a Web client receives a multiple choices response, it chooses one of the alternates and retrieves it from the specified location.

In this approach, however, Web clients must still contact the primary site to receive the list of alternates, and the administrator of the primary site must still manually create the list of alternate objects. This problem can be solved using "gossiping" or "epidemic" information dissemination algorithms [27]. We chose to use an architecture similar to the one used by NNTP Servers [28] to disseminate the list of mirrors to all mirror locations. This allows the client to use any mirror to obtain the full list of mirror locations for a particular object. In addition, the list of mirror locations can also be cached in the SPAND database.

A.2 Metrics for Mirror Ranking

The primary metric we use for the ranking of mirrors is the typical document download time as described in IV-B.1. In cases where a client experiences a failure in retrieving a document from a mirror, we report a large constant value which is greater than the worst possible download time.

A.3 Algorithm for Mirror Selection

Once a client has the list of mirror locations, it must select which location to download from. Regardless of the metric used to choose the best mirror, our algorithm must have the following features:

- Recommend efficient mirror locations
- Avoid overloading a single mirror location
- Ensure that the network performance information for lowly ranked mirror locations is kept up-to-date

These goals can sometimes conflict with each other. For example, to keep timely performance information for lowly ranked mirror locations, it is necessary to occasionally direct clients to these mirrors.

To achieve these goals, our algorithms generate the following four values:

- A primary location: The server that the client should use.
- A backup location: The best performing server.
- A target number of bytes: A performance level that should be considered acceptable for the primary location.

The system uses a weighted randomized selection for the primary location. For each mirror location, we use the response time x_i to construct a weight for that mirror w_i. We experimented with following five weighting functions:

- Uniform: $w_i = 1$
- Inverse: $w_i = 1/x_i$
- Inverse Squared: $w_i = 1/{x_i}^2$
- Exponential: $w_i = e^{-x_i}$
- Exponential-Uniform Hybrid: $w_i = e^{-x_i}$ or $w_i = 1$ with probability z. This means that $z\%$ of selections follow a uniform distribution and $(1-z)\%$ of selections follow an exponential distribution.
- Hyperexponential: $w_i = e^{-e^{x_i}}$

After the weights of the different servers are calculated, the sum of the weights is normalized to one (i.e. $\Sigma_j w_j = 1$). We randomly select a mirror location according to the weights and recommend that mirror to the client. This distributes load across multiple mirrors and assures that lowly ranked mirrors are still visited occasionally. In Section V-C, we show how the choice of weighting function affects client performance.

Unfortunately, clients sent to lowly ranked mirrors will face longer retrieval times. Since this significantly reduces the incentive to use our system, LookingGlass provides a backup mirror location for clients to use. When the LookingGlass HTTP proxy receives this information, it starts downloading the content from the primary location. If the proxy has not received the target number of bytes from the primary location at the end of the experiment time, LookingGlass switches to the backup location to complete the transfer. The proxy always generates a performance report for the primary mirror and the backup mirror if it switches to it.

The target number of bytes is initially fixed at a fraction (we used 10%) of the total document size. The system sets the experiment time to its estimate of how long it should take for the best known mirror to transfer that fraction of the document. Our algorithm also divides the target number of bytes by a constant called the *aggressiveness factor* before returning it to LookingGlass. This constant can be any value greater than 1. By changing the value of the aggressiveness factor, we make clients more or less aggressive in giving up on a lowly ranked mirror and switching to the backup location. Greater values of the aggressiveness factor make clients less likely to switch to a backup because the target number of bytes decreases. In Section V-C, we show how the choice of this constant affects client performance.

This algorithm has the attractive property that it bounds the amount of time that a client will attempt to use a possibly lowly ranked mirror before giving up on it and switching to the backup mirror. This meets both goals of maintaining reasonable client performance and keeping up-to-date network performance statistics.

B. *Experimental Methodology*

To test the effectiveness of LookingGlass, we analyzed a set application-level client robots download Web objects from a collection of Web servers that mirrored the same content. The clients downloaded objects in a series of rounds. In each round, the client retrieved the same object from each mirror location and recorded the document download time for each mirror. The results we present here are from a UC Berkeley client that downloaded a 450 Kbyte image from the Mars Pathfinder mission from a collection of 20 mirror locations. The primary mirror location was at the Jet Propulsion Laboratory, and the mirror locations ranged from the Bay Area to Hawaii.

To conduct our experiments, we used the traces to indicate what the client performance would be if it used LookingGlass to select the most appropriate mirror. At the beginning of each round, the client used LookingGlass to determine the primary and backup mirror locations, the experiment time, and the target number of bytes. We used the information from the trace to determine how long the client would actually have taken to complete the transfer, including downloading the document from the primary mirror and, if necessary, switching to the backup mirror. Only download times for the primary and (if the client switched) the backup mirror locations were added to the collection of performance information for the next round. For each round, we reported the ratio of the actual time taken by the client and the minimum observed download time for all mirrors in that round.

C. *Results*

In this section, we examine the sensitivity of our system to several parameters:

- The choice of weighting function in ranking mirror locations
- The scaling constant for the experiment time that controls the aggressiveness of clients in giving up on mirror locations

We also compare our system's use of network performance information against alternate methods, such as selecting the geographically closest, network topologically closest, random or primary mirror site.

C.1 Choice of Weighting Function

Figure 7 shows the results of using different weighting functions in ranking mirror locations. The aggressiveness factor was set to 5 in this experiment.

We see that the hyperexponential and uniform weighting functions lead to the worst performance, as they give too much and too little weight, respectively, to the poorly performing hosts. The uniform weighting function does not take any performance information into account, and clients often visit mirror locations that have bad performance. The hyperexponential weighting function assigns too much weight to hosts that are initially highly ranked. As a result, clients continually visit the same small set of mirrors and do not discover other mirrors that may have better performance.

The $1/x$ weighting function performs better than the uniform and hyperexponential functions but worse than the exponential and $1/x^2$ functions. As expected, the hybrid function ranges between the exponential and uniform functions depending on the value of the mixing constant z.

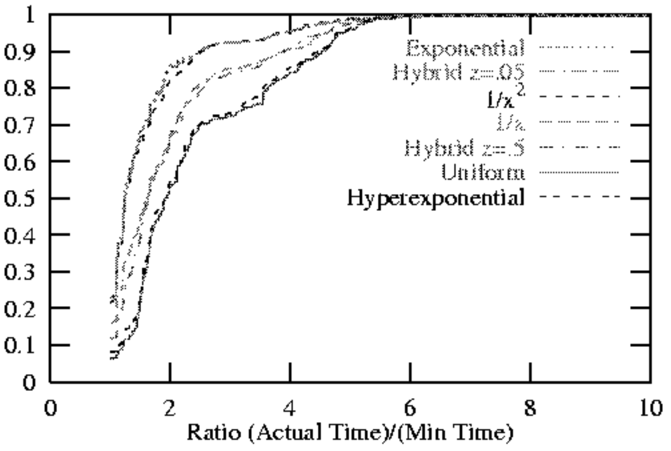

Fig. 7. Effect of choice of weighting function (from left to right: Exponential, Hybrid $z = .05$, $1/x^2$, $1/x$, Hybrid $z = .5$, Uniform, and Hyperexponential) in ranking mirror locations.

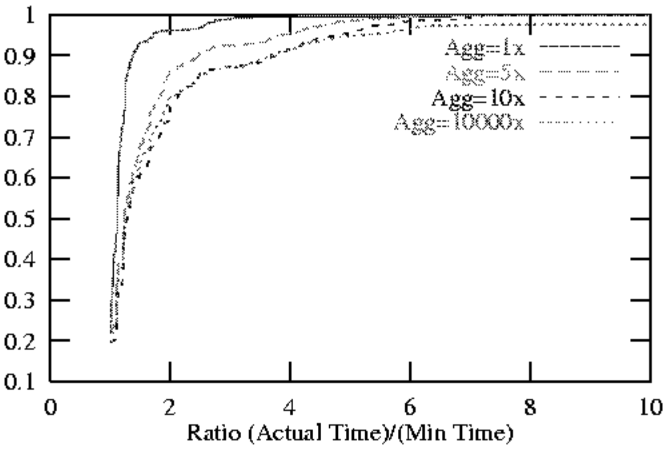

Fig. 8. Effect of aggressiveness factor on client performance. The factor increases from left to right curve.

C.2 Choice of Aggressiveness Factor

Figure 8 shows the sensitivity of the aggressiveness factor on client performance. The weighting function was a exponential-uniform hybrid with a 5% probability of choosing a uniform distribution for this experiment. As the factor decreases, clients become more aggressive, switching to the backup host even for small differences in performance. As the factor increases, clients become more tolerant of significant performance differences and as a result their performance degrades.

C.3 Choice of Ranking Metric on Performance

Figure 9 summarizes the maximum benefits of using LookingGlass over alternate ranking metrics that do not take network performance information into account. There are several CDF curves, each representing a different server selection method. In this experiment, LookingGlass used the exponential-uniform hybrid weighting function with a 5% probability of choosing a

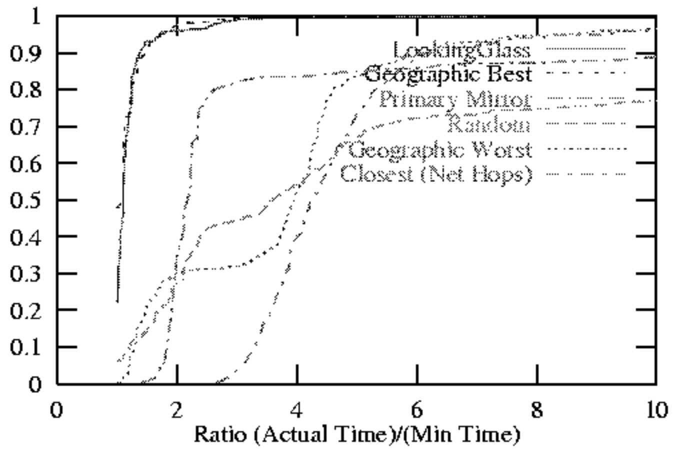

Fig. 9. Effect of choice of ranking metric on performance. From left to right (measured at y=.4): LookingGlass, Geographic Best, Primary Mirror, Random, Geographic Worst, and Closest.

- Geographic: Clients chose a mirror in the closest geographically region.
- Network Hops: Clients chose a mirror that was the fewest network hops away.
- Random: Clients select a different mirror for each round at random.
- Primary: Clients always use the primary location (JPL).

Because there were several mirror locations in the same geographic area as the client, we show the performance for two mirror locations in the Bay Area: one that performed well and one that did not.

We see that Looking Glass performs as well as or better than all other choices of ranking metrics. The network hops metric actually performs the worst of all the metrics. The Random approach performs worse than if clients simply contacted the primary site. Using geographic information may or may not lead to good performance. One Bay Area mirror performs as well as Looking Glass, while another performs much more poorly.

In contrast, LookingGlass performs very close to optimal, as a large fraction of the ratios are close to one. For 90% of the rounds, the download time using LookingGlass is within 37.5% of the minimum possible download time for that round (i.e. the ratio is 1.375 or less).

VI. Conclusions

In this paper, we presented a network measurement architecture that facilitates the development of adaptive applications. We described the assumptions and design choices that we made in developing our architecture and pointed out their advantages and disadvantages over alternate design choices. We described the core architecture and how its extensible nature makes it easy to add support for new application types. We also described how the architecture was used for a generic bulk transfer transport and an HTTP-specific transport. To validate our architecture, we presented LookingGlass, an HTTP Server Selection application that uses our architecture to help Web clients choose be- to download mirrored Web objects in near-optimal time.

References

[1] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, M. F Schwartz, and D. Wessels, "Harvest: A scalable, customizable discovery and access system," Tech. Rep. CU-CS-732-94, Computer Science Department, University of Colorado, Mar. 1995.
[2] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer, "Cluster-based Scalable Network Services," in *Proc. 16th ACM Symposium on Operating Systems Principles*, October 1997.
[3] E. Amir, S. McCanne, and H. Zhang, "An Application Level Video Gateway," in *Proc. ACM Multimedia*, Nov. 1995.
[4] S. Seshan, M. Stemm, and R. H Katz, "SPAND: Shared Passive Network Performance Discovery," in *Proc. Usenix Symposium on Internet Technologies and Systems (USITS '97)*, Dec 1997.
[5] R. L. Carter and M. E. Crovella, "Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks," Tech. Rep. BU-CS-96-007, Computer Science Department, Boston University, Mar. 1996.
[6] V. Paxson, *Measurements and Analysis of End-to-End Internet Dynamics*, Ph.D. thesis, U. C. Berkeley, May 1997.
[7] V Jacobson, "pathchar – A Tool to Infer Characteristics of Internet Paths," ftp://ee.lbl.gov/pathchar, 1997.
[8] "Cisco Distributed Director Web Page," http://www.cisco.com/warp/public/751/distdir/, 1997.
[9] J. Guyton and M. Schwartz, "Locating Nearby Copies of Replicated Internet Servers," in *Proc. SIGCOMM '95*, Sept. 1995.
[10] J. Gwertzman and M. Seltzer, "The Case for Geographical Push-Caching," in *Proc. Fifth IEEE Workshop on Hot Topics in Operating Systems*, May 1995.
[11] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin, "An Architecture for a GLobal Internet Host Distance Estimation Service," in *Proc Infocom '99*, 1999.
[12] "Sonar home page," http://www.netlib.org/utk/projects/sonar, 1999.
[13] P. Barford and M. Crovella, "Measuring Web Performance in the Wide Area," in *Performance Evaluation Review*, August 1999.
[14] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar, "A novel server selection technique for improving the response time of a replicated service," in *Proc Infocom '98*, Mar. 1998.
[15] R. Wolski, "Dynamically forecasting network performance to support dynamic scheduling using the network weather service," in *Proc. 6th High-Performance Distributed Computing Conference*, Aug. 1997.
[16] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven Layered Multicast," in *Proc ACM SIGCOMM*, Aug. 1996.
[17] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," in *Proc. Winter '93 USENIX Conference*, San Diego, CA, Jan. 1993.
[18] T. Von Eicken, D. Culler, S. C. Goldstein, and K. E. Schauer, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proc. International Symposium on Computer Architecture*, 1992.
[19] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review*, vol. 26, no. 2, pp. 5–18, April 1996.
[20] M. Mathis and M. Allman, "Empirical Bulk Transfer Capacity," http://www.ietf.org/internet-drafts/draft-ietf-ippm-btc-framework-00.txt, Jan. 1999.
[21] G.R. Wright and W. R. Stevens, *TCP/IP Illustrated, Volume 2*, Addison-Wesley, Reading, MA, Jan 1995.
[22] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "TCP Throughput: A Simple Model and its Emperical Validation," in *Proc. Sigcomm '98*, Aug. 1998.
[23] M. Mathis, J. Semke, J. Mahdavi, and Ott. T, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," *Computer Communications Review*, vol. 27, no. 3, July 1997.
[24] "Muffin Home Page," http://muffin.doit.org, 1999, Muffin Home Page.
[25] S. Seshan, M. Stemm, and R. H Katz, "Benefits of Transparent Content Negotiation in HTTP," in *Proc. Global Internet Mini Conference at Globecom 98*, Nov 1998.
[26] K. Holtman and A. Mutz, "Tranparent Content Negotiation in HTTP," http://genis.win.tue.nl/ koen/conneg/draft-ietf-http-negotiation-06.html, Jan. 1998.
[27] S.M. Hedetniemi, S.T. Hedetniemi, and A.L. Leistman, "A Survey of Gossping and Broadcasting in Communication Networks," *Networks*, vol. 18, pp. 419–435, 1988.
[28] B. Kantor and P. Lapsley, *Network News Transfer Protocol (NNTP)*, UC San Diego, Feb 1986, RFC-977.
[29] A. Myers, P. Dinda, and H. Zhang, "Characteristics of Mirror Servers on