Collecting Network Status Information for Network-Aware Applications

Nancy Miller and Peter Steenkiste
School of Computer Science
Carnegie Mellon University
e-mail: nam@cs.cmu.edu - prs@cs.cmu.edu

Abstract—
Network-aware applications, i.e., applications that adapt to network conditions in an application specific way, need both static and dynamic information about the network to be able to adapt intelligently to network conditions. The CMU Remos interface gives applications access to a wide range of information in a network-independent fashion. Remos uses a logical topology to capture the network information that is relevant to applications in a concise way. However, collecting this information efficiently is challenging for several reasons: networks use diverse technologies and can be very large (Internet); applications need diverse information; and network managers might have concerns about leaking confidential information. In this paper we present an architecture for a hierarchical collector of network information. The decentralized architecture relies on data collectors that collect information on individual subnets; data collectors can collect information in manner that is appropriate for that subnet and can control the distribution of the information. For application queries that involve multiple subnets, we use a set of master collectors to partition requests and distribute subrequests to individual data collectors and to combines the results. Collectors cache recent network information to improve efficiency and responsiveness. This paper presents and justifies the collector architecture, describes a prototype implementation, and presents preliminary measurements characterizing its operation.

Keywords—**Network-aware applications, Network monitoring.**

I. Introduction

In recent years we have seen a significant increase in the number of distributed applications, such as streaming of video and audio, remote data access, or distributed computations. As these applications use the network more aggressively, their performance becomes more sensitive to the level of service that the network can deliver. Such applications benefit from being "network-aware", i.e., they can adapt in an intelligent way to network conditions. For example, a video streaming application may adapt its frame rate, a user of remote storage may select the server with the best network connectivity, while a distributed computation may consider network bandwidth when doing node selection.

In order to adapt appropriately, network-aware applications need information about the network. While some applications can easily make use of *implicit* network feedback, e.g., packet loss is a sign of congestion, it has limitations. For example, implicit feedback only gives applications information about the part of the network they are using (making it hard to consider different options "in space") and it is not useful for decision making at start up, before execution has started. While applications can overcome these problems by probing the network using benchmarks, this is cumbersome and introduces overhead for both the application and the network.

The CMU Remulac project is proposing instead to provide applications with *explicit* information about network conditions using the Remos applications programming interface (API). Applications specify what information they need (e.g., delay or bandwidth) and the nodes they are interested in, and Remos will return estimates for the information. An approach based on a Remos-like API has several advantages compared with, for example, having applications probe the network. First, it is potentially more efficient since multiple users can share the same information. Second, since Remos presents the information in a network-independent fashion, applications developers can be less concerned about the details of the underlying network. Finally, it simplifies the task of the application developer since the task of collecting network information has been moved to network developers.

However, collecting explicit network information is challenging. Applications need very diverse information, raising the question of how to formulate a rich set of information without making the API overly complex. Moreover, it is not clear how accurate network performance information can be, given the dynamic nature of the network, and how accurate it should be, to be useful to applications. A final challenge is the complexity of satisfying diverse application requests in an efficient and scalable manner. In this paper we focus on the last question: we describe the design and implementation of a set of hierarchical "collectors" that collect raw network information that is used by the Remos API to satisfy application queries.

The remainder of this paper is organized as follows. We first give a brief overview of the Remos interface. We then discuss the design criteria for the Remos implementation and present a high level architecture. We then present a more detailed description of a prototype implementation and we use a set of measurements to characterize its functionality and performance. We conclude with a comparison with related work and a summary.

II. Remos

A Remos query specifies a set of endpoints and the type of network information that is of interest. Remos returns the requested information in the form of a *logical topology*, a set of switch nodes and links that are annotated with bandwidth or latency information. The Remos logical topology is typically much simpler than the actual network topology since network details that that do not affect application performance are omitted. For example, a sequence of links separated by routers may

This research was sponsored by the Defense Advanced Research Projects Agency monitored by Naval Command, Control and Ocean Surveillance Center (NCCOSC) under contract number N66001-96-C-8528 and by Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287.

be represented as a single (logical) link or a set of bridged Ethernets may be approximated by a single switch. Figure 1 shows an example of a logical topology (without annotations). Note that network managers would usually prefer the more detailed topology on the left, but the many low-level details would make it incomprehensible for most application developers or users.

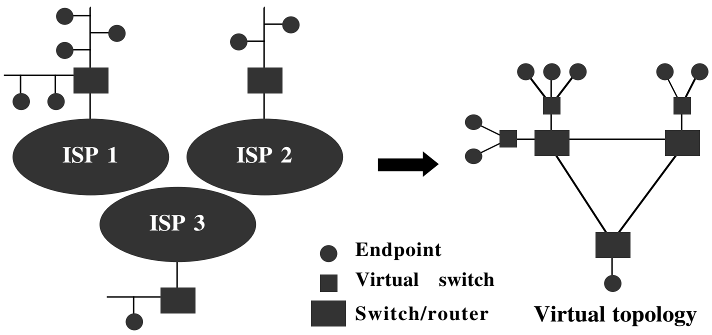

Fig. 1. Example of a logical topology

The logical topology is an attractive format to represent network information for applications. A first benefit is that it is more concise than alternatives such as listing bandwidth or latency information for each pair of endpoints, or the real topology. Second, it shows where application flows could be competing with each other for bandwidth. Finally, more complex application queries such as "find the 5 nodes with the best network connectivity" can easily be done using the logical topology [1], but are difficult to resolve using, for example, pair-wise information. Note that applications can obtain pair-wise information by issuing separate queries for each pair of endpoints. More details on the Remos API can be found elsewhere [2], [1], [3].

III. REMOS DESIGN

We present the design of the CMU Remos prototype.

A. *High-level Remos architecture*

Our Remos implementation has two layers: a collector and a modeler (Figure 2); they are responsible for network-oriented and application-oriented functionality, respectively. The collector layer is responsible for collecting basic static and dynamic network information that is relevant to applications. We describe the collector in more detail in the rest of this paper. The modeler is a library that can be linked with applications. It satisfies application requests based on the network information provided by the collector. The primary tasks of the modeler are as follows: generating a logical topology based on the information provided by the collector, associating appropriate information with each of the network components, and satisfying flow requests based on the logical topology. The modeler exports Remos information through Java and C interfaces.

B. *Collector design criteria*

The design of the collector has to meet the following requirements:

- *diverse information*: applications appear to be primarily interested in bandwidth information on different timescales although latency is sometimes also useful. Both static and dynamic information is of value.
- *network diversity*: the collector must be able to deal with a diverse networks. Networks can differ in terms of technology (e.g., ATM versus IP over SONET), but more importantly, they can differ in how network information can best be collected.
- *efficiency and responsiveness*: although explicitly collecting network information is often expensive, Remos queries should typically be fast and inexpensive.
- *scalability*: the collector implementation should be scalable in two ways. It should be able to handle reasonably large queries, although we expect that most users will be interested in a relatively small number of endpoints. A more important scalability parameter is that it should be able to handle a high rate of queries. This means that each query should be handled efficiently.
- *privacy*: some network managers (e.g., ISPs) may not be willing to expose all the details of their network. There might also be concern that having applications react to information that is very dynamic can create oscillation. To avoid these issues, it should be possible for network managers to control what information about their network is exported.

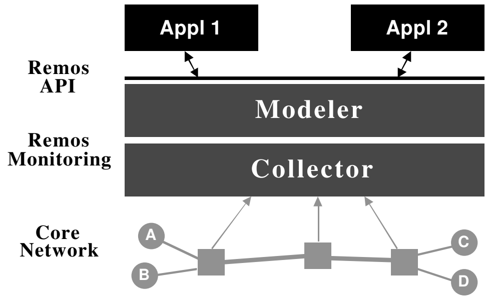

Fig. 2. Remos architecture

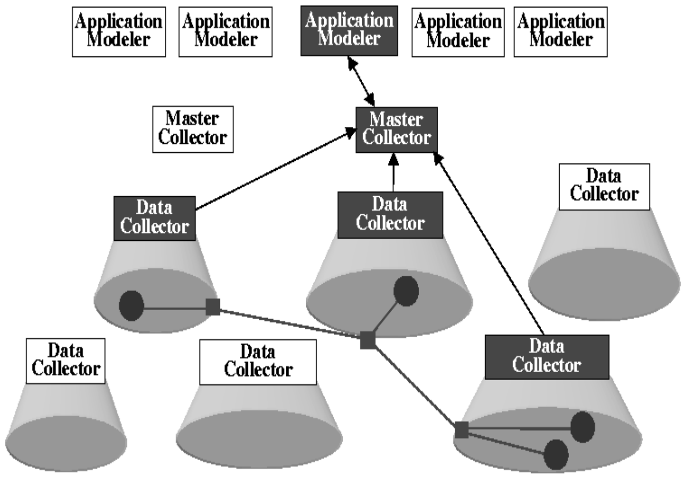

Fig. 3. Collector architecture

C. Collector architecture

Figure 3 shows the collector architecture. It has two types of components. First, there is a *data collector* associated with each individual domain, typically a single subnet or administrative domain. Each data collector is responsible for collecting information in its domain, and queries that involve only endpoints in a single domain can be handled completely by its collector. Queries that involve hosts in multiple domains, such as the four node example shown in the figure, are resolved using a *master collector*. The master collector identifies what data collectors are relevant to the query (using the host information), contacts those collectors with an appropriate query, and combines the information they return into a coherent result that is returned to the modeler.

The architecture of Figure 3 is driven by the requirements listed in the previous section. By having a collector per subnet, each collector can use methods of collecting network information that are appropriate for that subnet; we present examples in the next section. The administrator for that subnet can also control what information about the subnet is exported to internal and external users. Explicitly collecting network information is often expensive, and if every application query would require Remos to go out and collect the relevant information, the system would be of limited use since it would be slow (long response times) and resource intensive. To (on average) reduce both the cost and elapsed time of queries, data collectors cache information so they can reuse any information that was recently obtained. The architecture in Figure 3 lends itself well to the reuse of cached information. Data collectors will accumulate information about the domain for which they are responsible. We expect that master collectors will also be associated with a particular domain, and they will accumulate information both about that domain, and its connectivity to other networks. The scalability requirement is addressed in a variety of ways. First, we have multiple data collectors that are each responsible for a restricted local environment, plus a set of master collectors that operate at the inter-domain level. Note that this organization is similar to that of routing protocols. Moreover, the multiple master and data collectors can resolve queries in parallel. Finally, the efficiency gains from caching also improve throughput. The final requirement, information diversity, relates to the details of collector and master collector implementation.

An interesting question from the application perspective is how up to date and accurate information should be to be useful. Since there are relatively few network-aware applications, this is largely an open question, and it is likely that different applications will have very different requirements. The above architecture deals with this by having applications specify an *age* for their request. The age parameter, specified in seconds, indicates how old the information can be and it is used by the collectors to determine whether requests can be served from the cache. Applications that do not need very recent or accurate data can specify a large age value, which will allow the collector to return older data from its cache. Applications that need more up to date information can specify a smaller age value, but they may have to wait longer for a response if the collector is forced to refresh cached information.

The flip side of this issue is how closely the above architecture tracks changes in the network conditions. The tradeoff is clear: by probing the network more frequently, changing conditions can be tracked more accurately, but more resources have to be invested. In practice, there is a limit to how closely it is feasible to track changes, both because applications often cannot use very finely detailed information, and because of concerns about network oscillations, similar to the behavior sometimes seen with dynamic routing. Clearly, the above architecture allows managers to limit how frequently dynamic information is updated, thus limiting overhead and oscillation risks. For frequently requested information, it may in fact be advantageous to periodically update dynamic information, independent from specific queries.

D. The collector as a network service

We look at the collector not as a part of the core network structure, but as a value-added service provided by the network to its users. The collector sits on top of (outside) the core network infrastructure and should thus not slow down or add complexity to the its operation.

One aspect of the architecture described above is that modelers (applications) must locate the appropriate master collector or data collector, and the master collectors must be able to locate the data collectors they need. When viewed as services, both data collectors and master collectors can be discovered using a general service location protocol (SLP). A modeler could discover a master collector running in its subnet using a local SLP [4] and the master collector could use a wide-area SLP [5] to locate data collectors. The latter is harder, not only because there is a broader space to search, but also because the master collector has to determine what data collectors are involved and what subnets they are responsible for. However, SLP tags can be used to identify the subnet and scope (e.g., netmask) for each collector. An alternative would be to develop a separate discovery protocol for collectors (peer based protocol). However, the use of a general SLP, if available, seems preferable.

IV. The Remos prototype

The CMU Remulac group has implemented a Remos prototype based on the architecture described in the previous section. In this section we give a more detailed description of two different data collectors, the master collector, and the protocol they use to exchange network information.

A. Data collectors

Given a set of endpoints for a subnet, the task for the data collector is to collect topology information and dynamic performance features such as latencies and available bandwidth for the part of the network that is relevant to the endpoints. In this section we describe a collector that is based on SNMP and a collector that uses benchmarks. Work is in progress on other collectors, including a collector for 802-bridged networks.

A.1 SNMP collector

The SNMP collector is used in a routed environment in which IP routers and endpoints support SNMP [6]. The SNMP collector performs the following tasks:

- Topology discovery: the collector uses SNMP route queries to discover the path between the endpoints listed in the query. Using this information, the collector can construct the topology of the part of the network that is relevant to the query.
- The collector uses SNMP queries to collect information on each of the links in the topology. Static bandwidth information corresponds to the link capacity. Dynamic bandwidth information is obtained by sampling the counters that store accumulated bytes sent over the link. The current implementation assumes a fixed per-hop delay; programs such as ping could be used to obtain more accurate estimates.
- The collector virtualizes the topology, e.g., it adds logical switches where needed. Other virtualization tasks (e.g., collapsing a set of links into a single logical link) are left for the modeler since the master collector may need the more detailed information to merge information from different data collectors.

```
/* Find the path between all node pairs */

foreach pair of endpoints {
  currentnode = pair.source;
  do {
    if (path_known(currentnode, pair.destination)) {
      break;
    }
    nexthop = find_nexthop(currentnode,
                           pair.destination);
    currentnode = nexthop;
  } while (!same_subnet(currentnode, nexthop));
}

/*
  Use SNMP route lookup to find the next hop.  The
  information is stored by netmask, so we search one
  netmask at a time, starting with the longest one.
*/

find_nexthop(currentnode, dest) {
  route_found = false;
  foreach netmask in currentnode.route_table {
    possible_route = apply_netmask(dest, netmask);
    if (in_table(possible_route, netmask.table)) {
      route_found = true;
      nexthop = possible_route.nexthop;
      break;
    }
  }
  if (route_found) return nexthop;
  else return default_route.nexthop;
}
```

Fig. 4. Topology discovery in the SNMP collector

The most complex step is the topology discovery (Figure 4). While simple, the algorithm is quite expensive since it has a running time of $D \times N^2$, where D is the diameter of the network and N is the number of endpoints in the request. However, to help reduce the actual running time, our algorithm stops a path search for a (source,destination) pair when we reach an earlier discovered path to the same destination. We are also working on other ways to speed up the algorithm. For example, in subnets where routes are symmetric, half the SNMP queries can be eliminated. Also, the elapsed time can be reduced by parallelizing the search using multiple threads. Once the static topology is known, dynamic link bandwidth information can be collected by simply querying the relevant routers using SNMP. This is relatively inexpensive, especially since queries can be issued in parallel.

As described in the previous section, the data collector caches information and tries to serve queries from the cache before collecting information, using the age parameter in the request to determine whether the cached data meets the application's needs. However, even without caching, collector performance can be acceptable [1], in part because the usage model of applications matches the collector structure. Applications often call Remos at startup time to help decide what endpoints to use, and then they periodically use Remos to check for changes. Given a cold cache, the first query will be slow since it involves topology discovery, but later queries will be fast since the collector only has to collect dynamic information. Our SNMP collector periodically updates the link bandwidth information for each of the links in its cache so that queries for cached information can be responded to immediately. The query interval and the granularity of the measurement are both set to a default of five seconds. This parameter can be changed by sending a command to the collector, but the query interval and granularity cannot currently be set independently of one another.

The algorithm we described in this section is the one used for single collector queries. An additional step is needed when the collector is called as part of a multi-collector request by a master collector, as we will describe when we present the master collector.

A.2 Benchmark collector

There are many networks (e.g., ISPs) that do not respond to our SNMP queries, making it necessary to gather information about them in other ways. For such networks, we developed a collector that uses user-level benchmarks to collect information. While it is certainly possible to collect some internal information about these networks (e.g., [7]), benchmarking is a simpler way of getting fairly complete information.

The benchmark collector is quite simple. If a request comes in for connectivity information between a set of nodes, the benchmark collector uses benchmarks to measure the bandwidth and latency of the nodes in a pair-wise fashion. For bandwidth we use Nettest [8] and for delay we use traceroute. The benchmark collector must have permission to run code on the endpoints since some benchmarks require that specific programs are run on the source or destination. The benchmark collector is also expensive: the algorithm is N^2 with a large constant (time to execute a benchmark). In practice we only use this collector as a "WAN collector", so endpoints correspond to subnets and N is in practice small (e.g., two or three). Also, in an environment with many Remos users, we would rely on caching to keep track of connectivity to a larger number of subnets. Another possibility is to make the period with which the information is updated proportional to N. This design is similar to what is used in Globus [9].

There are some interesting differences between the SNMP and the benchmark collector:

- The benchmark collector can add a considerable load to the network traffic (while it is probing). The SNMP collector on the other hand, adds very little traffic but places an additional load on the routers since they have to respond to SNMP queries.

- The benchmark collector measures user-level performance. In contrast, the SNMP collector collects historical data on bandwidth use, which then has to be translated into an estimate of how much bandwidth a new user can expect. While our results show that this is possible, more experience is needed to show how accurately this can be done across a range of networks.
- The information from the benchmark collector is less detailed. For example, suppose we have a three node query (nodes A, B, C). If benchmarks show that the A-C bandwidth is 4 Mbs and B-C is 5 Mbs, it is not possible to predict what the result would be if A-C and B-C stream data at the same time. An SNMP collector would return a logical topology that shows where the bottleneck is, i.e. whether it is shared between the two flows or not.

Overall, our experience indicates that SNMP collectors are less intrusive and provide more accurate information, although it is difficult to evaluate the impact of the SNMP queries on router performance.

B. Master collector

Queries involving a single subnet can be handled entirely by the data collector responsible for that subnet, but for broader queries we need a master collector to coordinate information collection.

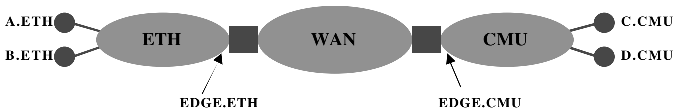

Fig. 5. Multi-collector example

We will use the example of Figure 5 when describing the functions of a master collector. It shows a network that connects four hosts, two at CMU and two at ETH in Zurich, Switzerland. A query sent to the master collector requesting information about these hosts might look like this:

```
REQUEST TOPOLOGY <age> LIST 4 A.ETH B.ETH C.CMU D.CMU
```

B.1 Identifying relevant subnets and collectors

For each query, the master collector has to identify the subnets that are relevant to the query, and which data collectors are responsible for those subnets. Relevant subnets fall into two categories. A first class includes subnets that host any of the endpoints listed in the request (CMU and ETH in our example). The second class includes subnets that do not host any of the endpoints but might carry some of the data between these endpoints, i.e., networks that provide connectivity between the subnets in the first class (e.g., WANs). In general, the first class will correspond to leaf networks belonging to users of the internet, while the second class includes ISPs that provide local or backbone internet connectivity.

The current implementation of the master collector uses a database to identify data collectors. The collectors register with the database, [1] giving information that includes the type of collector and the domain it is responsible for, represented by one or more subnet addresses and netmasks.

When it gets a query, the master collector first identifies the subnets in the first class, which can be done easily using the IP addresses of the endpoints. Once it has identified the subnets, it identifies the associated data collectors using the registration information described above. We will refer to these collectors as "LAN collectors"; in our current implementation all LAN collectors are realized as SNMP collectors. For the second class of subnets, the master collector relies on a global decentralized "WAN collector". It consists of a set of WAN collectors, one at each leaf network, that are responsible for characterizing the data paths from that leaf network to other subnets.

In our implementation WAN collectors are realized as benchmark collectors (Section IV-A.2); in this context, each WAN collector also provides a benchmark system that can be used by its peers at other sites. An (attractive) alternative is that ISPs would run their own collectors. In that case a WAN collector could discover what networks are involved in serving the query (e.g., using traceroute) and could then collect the necessary information by querying the collectors provided by the ISPs. While this approach may seem heavy-weight, caching averages the cost of collecting information over a large number of queries and the WAN collector design is in fact partly driven by caching considerations. Specifically, each WAN collector representative is responsible for WAN connectivity related to its subnet, which means that there should be a high degree of locality in the requests it receives.

Figure 6 shows pseudo code for the Remos master collector. It shows how the master collector finds all the LAN and WAN collectors that are responsible for the ip addresses given in the query (find_coll()). Since collector domains may overlap or there may be more than one collector registered for a domain, it prunes the list so that only one LAN collector and one WAN collector are contacted for each ip address (prune_collector_list()).

B.2 Breaking up requests and assembly of responses

Once the master collector has identified relevant data collectors, it has to formulate a query for each of them. This problem is trivial if we know the IP addresses of (the relevant ports on) edge routers that connect the subnets. However, this information is not part of the request. One solution is to have the master collector discover this information, e.g., using traceroute, or have it request the information from the data collectors. This approach has the drawback that the master collector has to learn about and keep track of information that is subnet specific. Instead, we place the responsibility for identifying edge routers with the data collectors. For each data collector, the master collector formulates a request that contains not only the endpoints of that subnet, but also one endpoint for each of the other leaf subnets. For our example, the master collector would create the following queries:

- For the LAN collector at ETH:
```
REQUEST TOPOLOGY <age> LIST 3 A.ETH B.ETH C.CMU
```
- For the LAN collector at CMU:
```
REQUEST TOPOLOGY <age> LIST 3 C.CMU D.CMU A.ETH
```
- For the WAN collector:
```
REQUEST ROUTE <age> A.ETH C.CMU
REQUEST ROUTE <age> C.CMU A.ETH
```

[1]The database is very similar to the directory agent in SLP, and we are in the process of moving towards using SLP.

```
/* Find relevant LAN and WAN collectors */
lancollectors = find_coll(ipaddrs, "LANCollector");
wancollectors = find_coll(ipaddrs, "WANCollector");
prune_collector_list(lancollectors);
prune_collector_list(wancollectors);

/* Formulate queries for WAN collectors */
foreach wancoll in wancollectors {
  querynum = 0;
  foreach wancoll2 in wancollectors {
    if (wancoll == wancoll2) continue;
    wancoll.query[querynum]
       = build_wan_query(wancoll2.iplist[0]);
    querynum++;
  }
  foreach ip in ipaddrs {
    if (is_covered_by_a_wancollector(ip)) continue;
    wancoll.query[querynum] = build_wan_query(ip);
    querynum++;
  }
  threadlist[numthreads]
       = create_thread_to_send_query(wancoll);
  numthreads++;
}

/* Formulate queries for LAN collectors */
foreach lancoll in lancollectors {
  lancoll.query = build_lan_query(lancoll.iplist);
  foreach lancoll2 in lancollectors {
    if (lancoll == lancoll2) continue;
    lancoll.query = add_to_query(lancoll.query,
                               lancoll2.iplist[0]);
  }
  foreach ip in ipaddrs {
    if (is_covered_by_a_lancollector(ip)) continue;
    lancoll.query = add_to_query(lancoll.query, ip);
  }
  threadlist[numthreads]
        = create_thread_to_send_query(wancoll);
  numthreads++;
}

/* Wait for data collectors results; merge results */
foreach thread in threadlist {
  wait_for_thread_to_finish();
  queryresults = merge(queryresults, thread.result);
}

queryresults = remove_duplicate_info(queryresults);
```

Fig. 6. Pseudo code master collector

The pseudo code in Figure 6 shows how both the requests for LAN and WAN collectors are built. For each WANcollector, it builds a query with a destination ip address from each other WANcollector's domain (build_wan_query()); that single IP address represents the subnet. For each LAN collector, a query is built containing all the ip addresses within that LAN collector's domain (build_lan_query()). Then the MasterCollector adds one ip address from each of the other LAN collector's subnets, and all the ip addresses that are not covered by any LAN collector (add_to_query()). This will ensure that each LAN collector reports all the appropriate edge routers in its domain that are along the path to the other domains. A separate thread is used to send each of the queries (create_thread_to_send_query()).

To handle requests from master collectors, we have to add a step to the data collector execution described in Section IV-A. Before it executes the steps described in Section IV-A.1, the SNMP collector has to replace the endpoints outside of its domain by the addresses of the edge routers that are used to get to and from that "foreign" endpoint. For example, the SNMP collector at CMU would have to replace A.ETH by EDGE.CMU. This information can easily be obtained using traceroute. A WAN collector will have to replace the endpoints outside its domain by the addresses of nodes in the same subdomains that can be used to start benchmark programs (e.g., a peer WAN collector).

```
HEREIS TOPOLOGY ENDOFTOPOLOGY
node C.CMU 128.2.185.90 compute -1.0
node D.CMU 128.2.185.91 compute -1.0
node c_v1CMU 10.128.2.1 vswitch 1250000
node EDGE.CMU 128.2.254.36 switch -1.0
edge EDGE.CMU 128.2.254.36 4 128.2.0.254
 128.2.211.254 128.2.254.36 198.32.224.64
link 128.2.185.90 10.128.2.1
undirected
1
1024352
1250000
link 128.2.185.91 10.128.2.1
undirected
1
1024352
1250000
link 128.2.254.36 10.128.2.1
undirected
1
1024352
1250000
ENDOFTOPOLOGY
```

Fig. 7. Example response from CMU SNMP collector

Once all the queries have been sent out, the master collector waits for each thread to finish reading the results from each of the collectors. Figure 7 shows an example of what a response might look like from the SNMP collector at CMU. Once all the data has been received, the master collector merges the results of each query (merge_results()) and removes duplicate node and link information (remove_duplicate_info()). In this process, node and link information from a LAN (SNMP) collector takes precedence over the same information from a WAN (benchmark) collector since the SNMP collector gives more accurate information about the links which fall inside its domain. In addition, if the response from an SNMP collector contains an edge router, it will also include a list of all of the IP addresses associated with that edge router. This is important because a router that is identified by one IP address in the response from a benchmark collector might be identified by a different IP address in a response from an SNMP collector. The master collector uses the edge router information

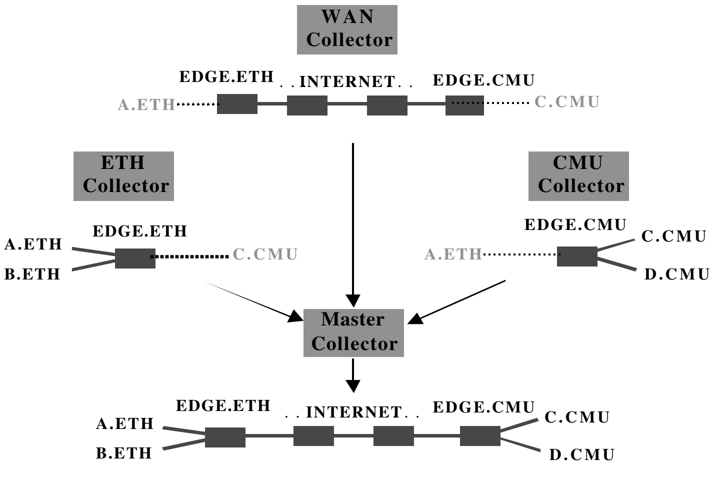

Fig. 8. Multi-collector example - merging of results

to determine that these two different ip addresses belong to the same node. The merging of different responses is illustrated at a high level in Figure 8.

C. The information request protocol

The information request protocol (IRP) allows the master collector or modeler to request information from a data collector. The main IRP request is the TOPOLOGY request. It specifies a set of nodes, represented by their IP addresses or hostnames, and returns a logical topology.

The logical topology is represented as a set of nodes and a set of links. There are three types of nodes: endpoints, switch nodes (routers), and logical switch nodes. The endpoints are the nodes listed in the original request, while the switch nodes are the routers or switches that are found during of the discovery of the routes between the endpoints. Only switch nodes that are relevant to the requests are included. The logical switch nodes are used to represent shared bandwidth pools, e.g., a shared ethernet segment. For example, when two or more endpoints are on a shared ethernet segment, a logical switch is used to represent that segment and the speed of the logical switch is set to the speed of the ethernet. Information returned about each node includes its hostname (if it has one), ip address, type, and speed. The speed of a node is an estimate of how fast it can transmit data (in the case of an endpoint) and how fast it can forward packets between interfaces (in the case of a switch). For each link, the IRP response lists what nodes it connects, the maximum speed of the link, the current available bandwidth on the link, and a latency value. Links are also specified as directed or undirected. An example TOPOLOGY query can be found in Section IV-B.

The caller can control the nature of the returned information through two parameters. One parameter controls the averaging interval for the information expressed in milliseconds (default 5000). This allows applications to match the coarseness of the information to their needs; i.e., applications that typically transmit large amounts of data need information on a coarser time scale than applications that burst smaller messages. The other parameter is an age parameter, which controls how old the static information is allowed to be. Specifying a non-zero value may allow the collector to return cached information which results in a faster response. While the protocol does not commit to a specific response time, users must understand that the age parameter can strongly affect the cost of a query.

The current implementation of the IRP was chosen to support convenient development and debugging rather than large scale deployment. Reliability is achieved by using TCP for the underlying transport. Requests and responses are in the form of ascii text, which is verbose but simplifies debugging. Requests take the form of an opcode followed by a set of parameters. Responses take the form of a list of “node” and “link” information elements. The main drawback of current approach is that it requires state in the collector for each client that it is communicating with (TCP connection state in the kernel and some minimal per-user state in the collector), which can limit scalability. At a later time we plan to replace the protocol with a more traditional request-response protocol (e.g., [10]).

D. Implementation and deployment

The master collector and data collectors are implemented in Java. One reason we use Java is that we envision that collectors could be downloaded and upgraded across the network, similar to the notion of Darwin delegates [11], [12] or active signaling [13]. The SNMP collector uses the Java SNMP agent library from AdventNet [14].

For performance reasons, each collector caches recent information. This will allow applications that use the same part of the network to share the burden of collecting the network information; alternatively, collectors can update frequently requested information periodically. The effectiveness of caching depends on the locality of the requested information. A data collector is responsible for collecting information for one subnet, leading naturally to high locality; large networks can be partitioned with multiple data collectors. We also expect each LAN subnet to run its own master collector and WAN collector. Both will accumulate information on the wide area connectivity for that subnet, leading again to high locality. More work is needed to refine the caching policies and to characterize their performance impact.

Except for some configuration information (e.g., the subnet they are responsible for), collectors only have soft state. The contact information for their peers and the network information is discovered dynamically. While there might be some performance benefit to keeping track of the network state current active requestors are interested in (as our current implementation does), this is an optimization and can also be done in a "soft state" fashion.

While the current prototype is rich enough to support applications and to gain initial usage experience, it is by no means complete. A complete implementation will need a richer set of data collectors, for example to deal with bridged networks [15] or ATM clouds, and we could also benefit from a richer set of tools to collect network information (e.g., pathchar, ..). It might also be possible to incorporate information collected from applications [16]. The current implementation does not limit access to network information, although the architecture lends itself to restricting the distribution of information, as described above. Also, we currently do not provide information on multicast communication, and we have not considered issues such as policy routing.

V. Experiments

We present the results of experiments demonstrating the Remos collector functionality.

A. Testbed experiments

We ran an experment on our private networking testbed to determine the accuracy of the SNMP collector. NetSpec was used to generate bursts of traffic of varying lengths between two endpoints on our testbed. The endpoints (233mhz Dec Alphas) were separated by two 233mhz routers running FreeBSD 2.2.6. Figure 9 compares the end-to-end bandwidth reported by Netspec with the bandwidth reported by the Remos SNMP collector measured in three second intervals. There is a good match between the results. Figure 10 shows Netspec bandwidth compared to the bandwidth reported by Remos at one second

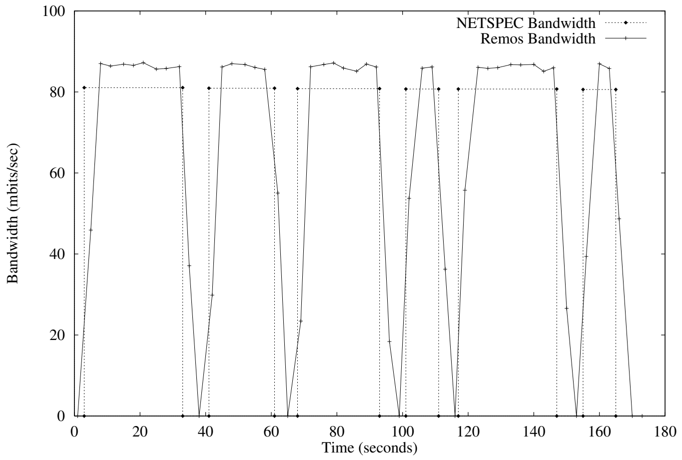

Fig. 9. Remos bandwidth measurements: 3 second interval

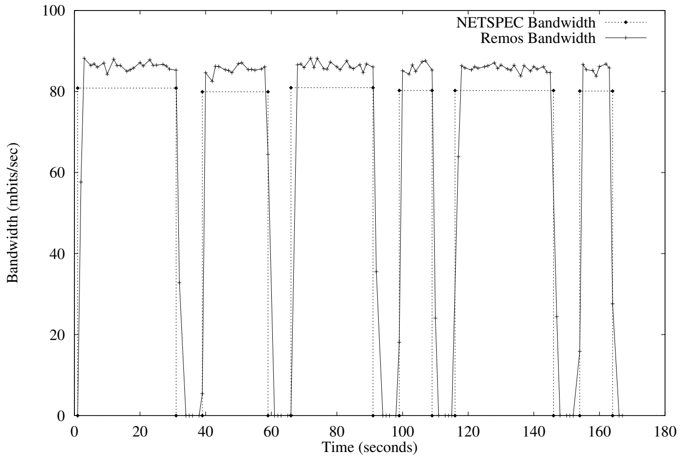

Fig. 10. Remos bandwidth measurements: 1 second interval

intervals. Remos tracks the application bandwidth more closely, at the expense of increasing the overhead on the router.

B. Timing experiments

Remos will only be useful to applications if it can respond to queries in a reasonable amount of time. As we discussed in Section III-C, the response time depends on how much of the data needed for the query is in the cache.

On our testbed, the SNMP collector was issued a query requesting a topology between a set of five hosts that were separated by four routers. When the collector had no data in its cache, it took an average of 52 seconds to return the topology over 5 trials. In Section IV-A.1 we discuss ways to improve the efficiency of the initial topology discovery, but even heavily optimized topology discovery will be relatively expensive. CLearly topology information is best collected periodically in the background so any application query will find at least the static network information cached.

When the collector has the topology cached and only has to collect dynamic information using the SNMP collector, the query took an average of 129 milliseconds over five trials, so SNMP collectors are fairly responsive.

C. Application experiments

One simple use of Remos is to help applications choose a remote server based on available network bandwidth. We have written a simple application that reads a file from a server after using network information obtained from Remos to choose the best server from a set of replicas. In our tests, we ran the application at Carnegie Mellon and servers at Harvard, ETH, and Argonne National Laboratories (ANL). Averaged over all experiments reported below, we observed an average throughput of 4.81 Mbs from ANL to CMU, 4.22 Mbs from Harvard, and 1.85 Mbs from ETH, so two of the servers have fairly similar performance characteristics.

In order to be able to evaluate the quality of the Remos information, we modified the application to read the file from all three servers, starting with the server that, according to Remos, has the best network connectivity. Over set of 230 experiements collected over a period of several days, Remos chose the remote site that ended up having the fastest transfer rate 66% of the time. In the other 34% of the trials, it chose the site that had the second fastest transfer rate. Given that the connections to two of the sites typically have similar performance, this is a good result. When Remos did not choose the best site, the file transfer rate from the one it did choose was only 0.77 megabits per second slower than the best site on average. Figure 11 provides some more details on the experiments.

```
When Remos chose the best site:
  Average difference in transfer rate between
    1st choice and 2nd choice: 1.10 mb/s
    1st choice and 3rd choice: 2.97 mb/s
When Remos chose the second best site:
  Average difference in transfer rate between
    1st choice and 2nd choice: 0.77 mb/s
    1st choice and 3rd choice: 2.50 mb/s
```

Fig. 11. Average differences in transfer rates

D. Prediction accuracy

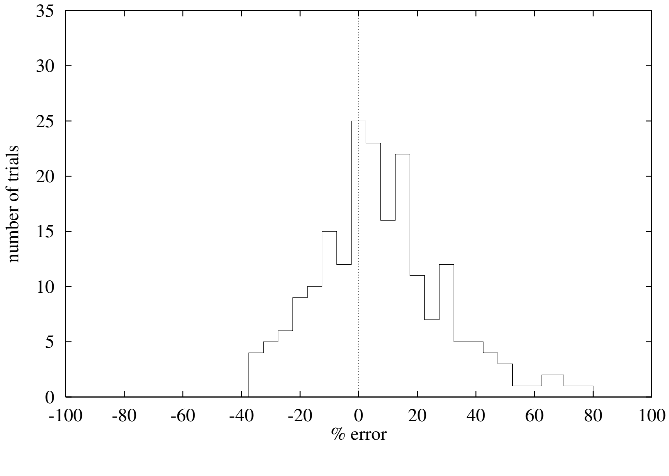

Fig. 12. Error histogram for throughput between ISI and CMU

Even though the focus of this paper is not on optimizing the accuracy of the network information, we present some measurements characterizing the accuracy of our measurements. We ran

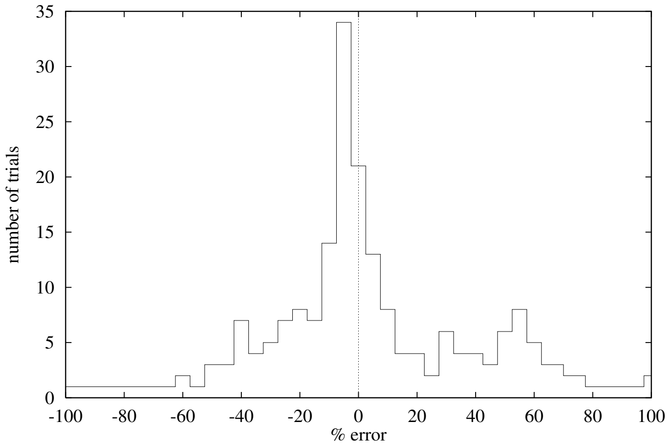

Fig. 13. Error histogram for throughput between Harvard and CMU

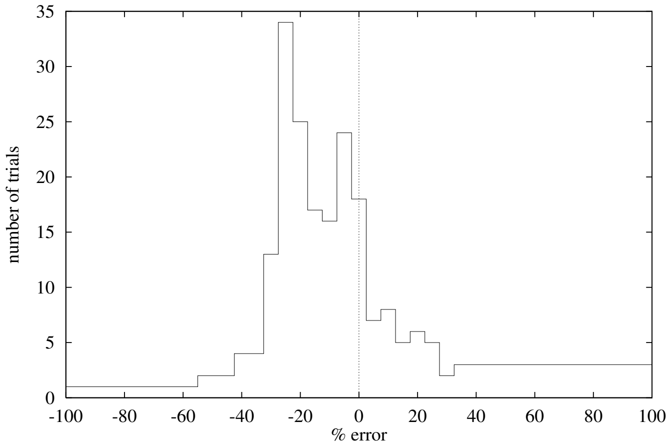

Fig. 15. Error histogram for Harvard with a 10M file size

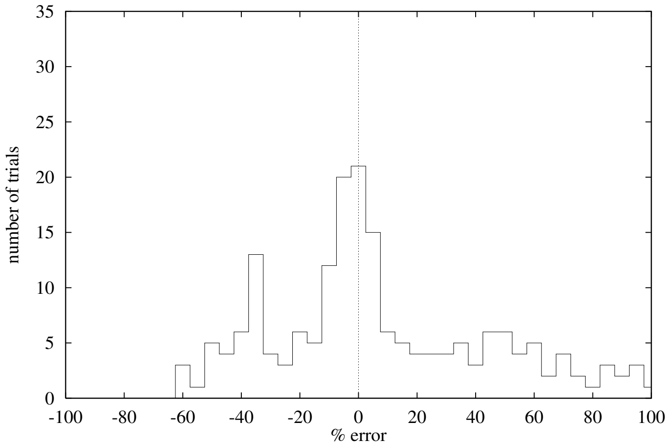

Fig. 14. Error histogram for Harvard with a 15 minute delay

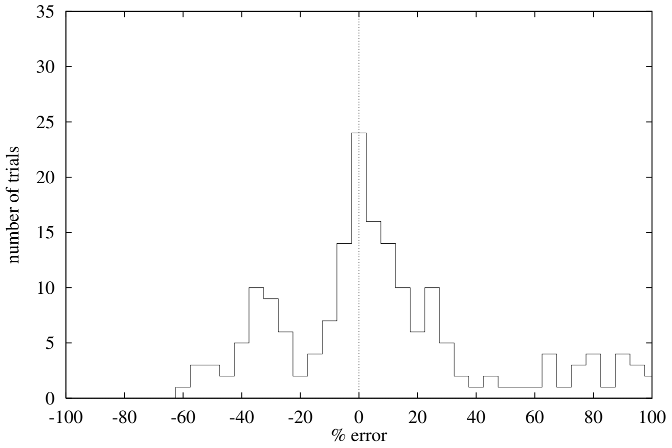

Fig. 16. Error histogram for Harvard with an 800K file size

a set of experiments between CMU and four sites (the same sites used earlier and ISI/USC in Los Angeles). The experiment is similar to the one described above, i.e., we ask Remos for an estimate of the available network bandwidth and we then use a simple file transfer to characterize the bandwidth an application would actually get. In these end-to-end experiments, the bottleneck is in the wide area, which is covered by our benchmark collector. It uses Nettest [8] to characterize throughput by streaming 1.6 MB of data between the two sites.

A first question is how accurately Remos can predict network throughput under ideal conditions, where Remos information is used immediately and the test that Remos uses to probe the network is very similar to the application. The experiment is simple. We first call Remos to characterize the bandwidth between CMU and one of the four remote sites, and then follow it immediately with a transfer of a file that is the same size of the Remos probe (1.6 MB). Note that this experiment is slightly different from the experiment in Section **??**, where each Remos query was followed by three file transfers with different delays relative to the probe. For each of the four sites we ran at least 190 trials. Figures 12 and 13 show a set of histograms that summarize the difference between the Remos throughput estimates and the application throughput. We observe that in this ideal case, the match is pretty good. The results we observed for ANL and ETH were also very similar to those shown here.

It is of course unrealistic to assume that Remos can always probe the network on demand using a benchmark that matches the requesting application. In the next set of experiments we change the experiment to model the case that an application receives stale information (e.g., information that was collected earlier by a periodic probe) or information based on a test that is different from the application. Figure 14 shows the difference between Remos throughput estimates and application throughput if the file transfer is performed 15 minutes after Remos executes its probe; both the probe and file transfer use 1.6 MB of data. Figures 15and 16 show the difference when the data sizes are different. The probe still uses a 1.6 MB transfer, but the application file transfer is for files of size 10 MB and 800 KB respectively. Not surprisingly, we see that the histograms are more spread out compared with the ideal case of Figure 13. However, overall the results are still good, and for many applications the information could be useful.

Clearly these measurements only scratch the surface of the problem of predicting networking information. Both Remos and other projects (e.g. [16]) have started to look at this question in more detail.

VI. Related work

Systems that focus on measurement of communication resources across internet-wide networks include Network Weather Service (NWS) [17] and *topology-d* [18]. NWS makes resource measurements to predict future resource availability, while *topology-d* computes the logical topology of a set of internet nodes. Both of these systems actively send messages to make communication measurements between pairs of computation nodes. Other examples include the network information discovery in Globus [9] and Prophet [19], which have similar features. An important characteristic that distinguishes Remos is that applications interact with a portable interface to the network that includes flow query and logical topology abstractions. Remos implementations collect information in different ways, including using SNMP queries which are more efficient. The hierchical architecture of the Remos collector should also make it more scalable.

There has also been research in the network community on characterizing network performance for applications. One example is the host distance estimation service project [20] that characterizes the "distance" between hosts for the applications. Both projects use a two-level hierarchy of collectors, although the breakup of responsibilities is slightly different. A more significant difference is that the network distance project focuses on static (minimal) delay and is coupled more closely to the core network, while Remos offers dynamic latency and bandwidth information and presents network information as a value-added service. These differences are in part the result of the fact that Remos is trying to support a class of more complex multipoint applications (e.g., distributed computing, filtering for video streaming, optimizing access to image databases).

There has also been work on characterizing the network for diagnostic purposes [7]. We believe that this work could be used to develop better collectors.

VII. Conclusion

Network-aware applications need information about network conditions to adapt, but collecting this information is challenging for several reasons, e.g., the scale and heterogeneity of today's networks. In this paper we present an architecture for a "collector" of network information. The architecture uses data collectors to collect information for individual subnets. Data collectors can use whatever methods are appropriate for their subnet to collect the information, and can place constraints on who gets access to what information. For application queries that involve multiple subnets, a master collector breaks up the application query and distributes it to the data collectors that are relevant to the query. It also collects and merges the results and responds to the user.

While the prototype is by no means a production system, it is complete enough that it can support applications, allowing us to gain application experience. We presented some preliminary measurements that show that our collector collects reasonable data for queries involving multiple subnets, and that simple applications can indeed make use of this to adapt to the network. While clearly much more work is needed, we believe that the proposed architecture is a good first step towards a general service that provides network information for applications.

VIII. Acknowledgements

We thank the members of the Remulac group for their contributions, especially Bruce Lowekamp and Dean Sutherland, who implemented the Remos modeler. We also thank Thomas Gross and Urs Hengartner for giving us access to systems at ETH, HT Kung for access to the Harvard system, the University Corporation for Atmospheric Research for giving us access to systems at NCAR, Carl Kesselman for access to systems at ISI, and the Argonne National Laboratory for giving us access to the systems at the High Performance Computing Research Facility, Mathematics and Computer Science Division.

References

[1] Bruce Lowekamp, Nancy Miller, Dean Sutherland, Thomas Gross, Peter Steenkiste, and Jaspal Subhlok, "A Resource Query Interface for Network-aware applications," *Cluster Computing*, , no. 2, pp. 139–151, 1999.

[2] Tony DeWitt, Thomas Gross, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, and Jaspal Subhlok, "ReMoS: A Resource Monitoring System for Network Aware Applications," Tech. Rep. CMU-CS-97-194, Carnegie Mellon University, December 1997.

[3] Bruce Lowekamp, Nancy Miller, Dean Sutherland, Thomas Gross, Peter Steenkiste, and Jaspal Subhlok, "A Resource Query Interface for Network-aware applications," in *7th IEEE Symposium on High-Performance Distributed Computing*. IEEE, July 1997.

[4] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan, "Service Location Protocol," Request for Comments 2165, June 1997.

[5] J. Rosenberg, H. Schulzrinne, and B. Suter, "Wide Area Network Service Location," work in progress, Internet Draft, draft-ietf-srvloc-wasrv-01.txt, November 1997.

[6] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser, "Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)," January 1999, RFC 1905.

[7] R. Siamwalla, S. Sharma, and S. Kashav, "Discovering internet topology," July 1998, http://thelonious.cs.cornell.edu/skeshav/papers/discovery.pdf.

[8] Cray Research Inc., "Nettest Networking Benchmark," ftp://ftp.sgi.com/sgi/src/nettest.

[9] Ian Foster and Carl Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal on Supercomputing Applications*, vol. 11, no. 2, 1997.

[10] Keith Moore, "SONAR - A Network Proximity Server - Version 1," work in progress, Internet Draft, draft-moore-sonar-03.txt, August 1998.

[11] Prashant Chandra, Allan Fisher, Corey Kosak, T. S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, and Hui Zhang, "Darwin: Customizable Resource Management for Value-Added Network Services," in *Sixth International Conference on Network Protocols*, Austin, October 1998, IEEE Computer Society.

[12] Eduardo Takahashi, Peter Steenkiste, Jun Gao, and Allan Fisher, "A Programming Interface for Network Resource Management," in *1999 IEEE Open Architectures and Network Programming (OPENARCH'99)*, New York, March 1999, IEEE.

[13] Bob Braden, "Active Reservation Protocol (ARP)," Dec. 1998, Abstract at URL http://www.isi.edu/div7/ARP/.

[14] Advent Net, "Java SNMP Agent Library," http://adventnet.com/.

[15] Bruce Lowekamp, dave O'Hallaron, and Thomas Gross, "Direct network queries for discovering network resource properties in a distributed environment," February 1999, submitted for publication.

[16] M. Stemm, S. Seshan, and R. Katz, "Spand: Shared passive network performance discovery," in *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, June 1997.

[17] R. Wolski, N. Spring, and C. Peterson, "Implementing a performance forecasting system for metacomputing: The network weather service," Tech. Rep. TR-CS97-540, University of California, San Diego, May 1997.

[18] K. Obraczka and G. Gheorghiu, "The performance of a service for network-aware applications," Tech. Rep. TR 97-660, Computer Science Department, University of Southern California, Oct 1997.

[19] John Weissman and Xin Zhao, "Scheduling Parallel Applications in Distributed Networks," *Cluster Computing*, vol. 1, no. 1, pp. 95–108, May 1998.

[20] Paul Francis, Sugih Jamin, Vern Paxson, Lixia Zhang, Daniel Gryniewicz, and Yixin Jin, "An architecture for a global internet host distance estimation service," in *Infocom'99*, New York, March 1999.