ReMoS: A Resource Monitoring System for Network-Aware Applications

Tony DeWitt Thomas Gross Bruce Lowekamp
Nancy Miller Peter Steenkiste Jaspal Subhlok
Dean Sutherland

Dec 97

CMU-CS-97-194

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Effort sponsored by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

Abstract

Network-aware applications can adjust their resource demands in response to changes in the availability of resources. Such applications must be able to obtain information about the status of the network resources. Providing such information to an application is conceptually simple, yet defining an interface that addresses the needs of applications, as well as the realities of current and future networks, is far from easy. The Remos interface described in this paper allows network-aware applications to obtain information about network capabilities and network status.

Interesting network information may be generated by the network hardware (switches), the network interface, or the network software, and is often in a system-specific format. Further, network architectures significantly differ in their ability to provide such information in a timely and accurate manner. Remos provides a standard interface format that is independent of the details of any particular type of network. When hosted on an advanced network architecture, Remos provides access to accurate status and capability information. On legacy networks, Remos provides as much information as is feasible, using best-effort approximation where appropriate. Using Remos, network-aware applications can be written independent of any particular network architecture, yet have the ability to tune their execution behavior to the dynamic state of the network.

In this paper we motivate and describe the Remos interface. We also sketch our first implementation of the interface for an IP-based testbed. Our experience in defining and implementing Remos indicates that providing accurate feedback in an network-independent manner is a significant challenge and we identify a number of areas for future work. The network-independent Remos interface establishes a framework that can form the basis for further research.

Keywords: Distributed computing, network computing, network-aware, system-aware, QOS, adaptive computing

1 Introduction

Networked systems provide an attractive platform for diverse applications ranging from scientific computing, collaborative design and interactive simulations, to distributed databases. The nodes of such a system may provide computing resources, memory, access to data collections, or sensor input or output devices. The challenge for an application developer is to utilize these resources while meeting constraints imposed by user demands (e.g., response time), application requirements (e.g., memory or input devices), reliability, and cost.

Effective use of the resources of a networked system is a major challenge for today's applications. Network conditions are continuously changing due to the sharing of resources. Congestion impacts connections by reducing the effective bandwidth and/or increasing the response time. On compute nodes, other users may start computations on shared nodes, and higher-priority activities may preempt user jobs. The CMU Remulac project is developing support for system-aware applications in the form of libraries that help applications retrieve system information and adapt. The Remos system, which is the subject of this paper, focuses on the network component, which can be the critical resource for many applications.

Two recent developments attempt to address the issues raised by the changes in resource availability in networks: (1) Quality-of-service (QoS) assurances by the network reduce the variability seen by the applications; (2) network-aware applications adjust to changes by modifying their resource demands. These two approaches are complementary. Services with a QoS assurance are more expensive than best-effort services, so applications may prefer to adjust rather than pay a higher price. Moreover, an application purely based on QoS assurance must be able to determine its resource demands in advance and limit resource usage to those provided by the QoS service, which is a significant restriction on flexibility. In general, it is important to be able to control the resource demands of applications so that resource consumption is in line with (observed or reserved) resource availability.

Network-aware applications must be able to obtain information about resource availability, in particular, the network's capabilities and status. Unfortunately, network architectures differ significantly in their ability to provide such information to a host or to an application executing on such a host. In order to avoid dependences on idiosyncrasies of network architectures and communication systems, application development for networks requires a system-independent interface between applications and networks. This allows the development of portable applications that can adapt on any network that can provide adequate information. Furthermore, a system-independent interface is crucial for allowing applications to exploit heterogeneous networks, where subnets are realized with different network technologies.

This document describes Remos, which represents our solution for high-level, portable interaction between a network and network-aware applications. Remos is a standard interface for applications to obtain the relevant network information needed for effective adaptation. We present the motivation and usage models for Remos, describe the Remos design, and illustrate how Remos is used with several examples. We also describe the ongoing Remos implementation on the CMCL networking testbed at Carnegie Mellon.

2 Usage models

The system we are targeting consists of *compute nodes* (or hosts) and *network nodes* (or switches), that are connected by physical communication links. The network nodes and links jointly form the network. The benefits of adaptability in such a system have been argued in a number of contexts [22, 4]. We summarize some typical scenarios that can exploit network status information.

- Node selection: When mapping an application onto a distributed system, it is necessary to determine the number of nodes (hosts) to be employed, and to select specific nodes for the computation. Many applications are developed so that they work with an arbitrary number of nodes, but increasing the number of nodes may drive up communication costs, while a certain minimum number of nodes are often required to fit the data sets. Example applications of this nature developed at Carnegie Mellon include Quake modeling [3] and Airshed pollution modeling [26].

- Optimization of communication: Closely related to the first issue is the problem of exploiting low-level system information, such as network topology. If an application relies heavily on broadcasts, some subnets (with a specific network architecture) may be better platforms than others.

- Load balancing: Changes in the environment may render the initial selection of nodes and connections unattractive. If the application has no flexibility with regard to the amount of communication and computation that must be performed, then the only option is to modify the mapping during execution to take the changes into account. The Fx programming environment at Carnegie Mellon [28] allows dynamic remapping of programs, although it does not currently have facilities to query its execution environment

- Application quality metrics: Some applications want to meet an application-specific quality model, e.g., jitter-free display of an image sequence, or synchronized delivery of audio and video signals. There may exist a variety of ways to comply with the demands, e.g., by trading off computation for communication. As the network environment changes, the application has to adjust its mix of communication and computation. (Note the difference to the previous scenario: in the above case, we change the nodes/connections that process or transmit data; in this case, we change the (amount or type of the) data that are transmitted. A fixed data source (sensor, film repository) may inhibit the first option. Practical situations may exhibit combinations of these two scenarios.)

- Function and data shipping: In some scenarios, a tradeoff is possible between performing a computation locally and performing the computation remotely, and such tradeoffs depend on the availability of network and compute capacity, based on a specific cost model. An example scenario is energy consumption in a battery-powered environment [22].

In the above usage examples, "system-aware" applications will in general have to be aware of all system resources, including both network and endpoint resources. The Remos interface focuses on providing application with information about networking resources only. We assume that applications can collect information about computation and memory resources on the network through another interface, possibly using a real-time operating systems running on the endpoints. However, for completeness, we have included a simple interface to system resources in Remos.

The Remos interface also does not support actual communication. Many communication libraries such as PVM [12] and MPI [10] provide a wealth of communication primitives, so there is no need to duplicate this aspect of distributed computing. In fact, as we will show later in the report, Remos could be used to optimize these libraries by customizing the implementation of group communication primitives for a particular network.

Finally, Remos provides a key ingredient for application-specific adaptation to network conditions, but does not solve the full problem. Information provided by Remos (or similar interfaces) can be used by applications to adapt, and this will significantly influence the design of such network-aware applications. A variety of projects are addressing the problem of intelligent network-based computing, including Remulac at Carnegie Mellon, Globus [11], and Legion [13]. Remos addresses a critical component of this problem.

3 Design challenges

In this section we discuss the problems that a common, portable interface like Remos must address. In the next section we present the design for the Remos system and explain how it addresses the challenges presented here.

3.1 Dynamic behavior

A first problem is that we have to characterize network properties that can change very quickly. Moreover, application traffic can have widely different characteristics, so applications may want access to different types of information. For many data intensive applications, the burst bandwidth available on a network may be more important than the average bandwidth. In contrast, applications that stream data on a continuous basis, such as video and audio applications, may be more interested in the available bandwidth averaged over a longer time interval. The Remos interface will have to satisfy these diverse requirements.

An application is most interested in the expected traffic on the network in the future. A preview of future properties would allow the application to adjust to the actual situation encountered in the next t units of time. Unfortunately, information on future availability of resources is impossible to find in general, although some sophisticated network management systems may be able to provide a good estimate based on the current knowledge of applications and their resource usage. This is an open research problem.

Finally, an application must be able to obtain information about changes in the network throughout its execution. An application may poll the Remos interface, or changes in the environment may trigger adjustments. Devising a software structure for the development of adaptive applications is still a topic of active research, so both interfaces should be supported.

3.2 Sharing

Connections (as seen by the applications) will, in general, share physical links with other connections of the same and other applications. This dynamic sharing of resources is the major reason for the variable network performance experienced by applications, and Remos will have to consider the sharing policy when estimating bandwidth availability.

An important class of applications that Remos attempts to support is parallel and distributed computations that simultaneously transfer data across multiple point-to-point connections. This kind of traffic pattern is typical of large-scale scientific computations (when mapped onto a distributed system) and distributed simulations. Since multiple parallel data transfers may be competing for the same resources, Remos will have to consider all exchanges in a data transfer step collectively rather than separately for each pair of endpoints. Again, this will require characterizing sharing behavior.

Determining a solution to the problem of characterizing the performance of multiple simultaneous data transfers is complex due to the interactions between the specific topology, network sharing policies, and the timing of messages.

3.3 Heterogeneity

Most installed networks do not have facilities that can directly provide the information applications need to adjust their resource demands. For that reason, highly portable, standard protocols such as TCP/IP rely on indirect mechanisms to obtain information about the network status, e.g., dropped packets indicate congestion. When one tries to collect more specific information about network conditions, heterogeneity becomes a major problem, both in terms of the diversity of the information that is of value to applications and diversity in networks.

The information that may be of interest to applications includes static topology information, routing information, dynamic bandwidth information (possibly averaged over different time intervals), and packet latency information. Different types of information are generated by different entities, are maintained in different formats and locations, and change on different time scales. For example, some information may only be available through a static database, e.g., maintained off-line by a system administrator. Other information may accessibly in a systematic fashion using protocols such as SNMP [6]. Finally, some information, e.g. packet latency, is not routinely collected and may have to be measured directly by the Remos system using benchmarks specifically developed for this purpose.

A second type of heterogeneity is that of the networks themselves: networks differ significantly in how much information they collect and make available. For example, dynamic link utilization of point-to-point links connecting routers can often be obtained through SNMP. However, shared Ethernets typically do not have a single entity collecting information on network utilization. Some networks do provide very specific feedback to endpoints on available network bandwidth as part of traffic management (flow control). The most important example is Available Bit Rate (ABR) traffic over ATM networks, where rate-based [5] or credit-based [7, 19] flow control tells each source how fast it can send. This information is currently only used at the ATM layer, but could be made available to higher-layer protocols or applications.

The challenge in designing a portable interface is to find a way to cover the entire range from currently deployed networks such as shared Ethernets with very large numbers of users, to more advanced commercial networks such as ATM. For an interface to be useful, it is not necessary that all information is available for every network. Sometimes partial information, such as static link capacities, may have significant value to applications. However, dealing with partial information is likely to make application development more complicated.

3.4 Level of abstraction

One of the thorny issues in designing an interface between applications and networks is to decide what aspects of the network should be exposed to applications. This problems shows up in a number of contexts.

One issue is heterogeneity. As discussed earlier, hiding network-specific details is important for portability, but it is not always clear how to do that without losing important information. A second example concerns limiting the volume of information provided to the application. Since all relevant hosts may be connected to the world-wide Internet, we need a way to limit the amount of information returned to an application, since providing information on the entire Internet is both unrealistic and undesirable. In some cases the scope of the query can be limited easily. For example, applications restricted to a LAN will only need information about the LAN. In other cases, solutions are less obvious. For example, how do we limit information for an application using three hosts, one located at Carnegie Mellon University, a second at ETH in Zurich, and a third on a plane flying from the US to Japan ?

Another important problem is management of routing information: if there are multiple paths between two hosts A and B, a network architecture may use different paths for individual data units (packets) traveling from A to B, and these paths may exhibit vastly different performance characteristics. Exposing this detail to the application is problematic since it is at a low level and changes rapidly. Furthermore, current protocols do not allow applications to influence routing, or even request that routing remains fixed. Therefore the fact that there are multiple physical paths, and that specific routing algorithms are responsible for performance differences and changes, is of limited value to applications. On the other hand, hiding the information is not without problems if the two paths differ in bandwidth and/or traffic.

All these problems center around the same question: what is the right level of abstraction for network information provided to the application. One option is to present the entire network topology, along with routing information and the characteristics of each link and node on the network. Such a description would provide all possible information, but it includes many details that are not relevant to most users of Remos, and it may be difficult to interpret. The other extreme is to provide the information at a much higher level, focusing on the performance characteristics that may be of interest to the application. This interface would be easier to use and the problem of information overload is avoided. However, in some situations, this may lead to information being vague and inaccurate, and potentially useless to applications.

4 Remos design

Remos is a query-based interface to the network state. Queries can be used to obtain the structure of the network environment, or to obtain information about specific sets of nodes and communication links on the network. The main features of the Remos interface can be summarized as follows:

Logical network topology: Remos supports queries about the structure of the network environment. The structure presented is a "logical topology", i.e., a representation of the network characteristics from an application's standpoint, which may be different from the physical network topology.

Flow-based queries: Queries regarding bandwidth and latency are supported for *flows*, which are logical communication channels between nodes. Flows represent application-level connections, and therefore, should be an easy to use abstraction for applications.

Multiple flow types: Remos supports queries relating to fixed flows with a fixed bandwidth requirement, variable flows that share bandwidth equally, and additional independent flows that can absorb all unused bandwidth.

Simultaneous queries: An application can make queries about multiple flows simultaneously. The Remos response will take any resource sharing by these flows into account.

Variable timescale queries: Queries may be made in the context of invariant physical capacities, measurements of dynamic properties averaged over a specified time window, or expectations of future availability of resources.

Statistical measures: Remos reports all quantities as a set of probabilistic quartile measures along with a measure of estimation accuracy. The reason is that dynamic measurements made by Remos typically exhibit significant variability, and the nature of this variability often does not correspond to a known distribution.

In the remainder of this section, we provide some details of the Remos interface, describe how the Remos features address design requirements stated in the previous section, and justify the important design decisions. We also illustrate the interface with simple examples and state the main limitations.

4.1 Query-based interface

Remos is a query-based interface and it supports flow-based queries as well as queries about the topology of the network environment.

Queries allow an application to specify what information it needs. An application is generally interested only in a subset of the existing nodes and in the performance of some communication operations. For example, an application that knows that broadcast operations are important for its performance may want to inquire about that particular capability. The advantage of this approach is that applications get precisely the information they need, and that the Remos implementation can use this context information to limit the work it needs to do to answer the query. The main alternative to a query-based interface is to provide applications access to a database of all network knowledge. However, this requires that a large amount of information must be constantly measured and updated, even when typically only a small fraction of it will be used.

Another mode of interaction between applications and network is via *callbacks*, where the network informs the application when the network behavior changes beyond application specified limits. Callbacks are attractive because they relieve the application from the burden of constant polling, and we plan to add a callback interface to Remos in the future.

4.2 Flow-based queries

A flow is an application level connection between a pair of computation nodes, and queries about bandwidth and latency on sets of flows form the core of the Remos interface. Using flows instead of physical links provides a level of abstraction that makes the interface portable and independent of system details. All information is presented in a network independent manner. While this provides a challenge for translating network specific information to a general form, it allows the application writer to write adaptive network applications that are independent of heterogeneity inherent in a network computing environment. We will discuss several of the important features of the flow-based query interface in the remainder of this section.

Multiple flow types

Applications can generate flows that cover a broad spectrum. Flow requirements can range from fixed and inherently low bandwidth needs (e.g. audio), to bursty higher bandwidth flows that are still constrained (e.g. video), to unconstrained flows that can consume any available bandwidth. Different flow types may require different types of queries. For example, for a fixed flow, an application may be primarily interested in whether the network can support it, while for an unrestricted flow, the application may want to know what average throughput it can expect in the near future.

Remos collapses this broad spectrum to three types of flows. A first type consists of *fixed flows* that can only use a limited amount of bandwidth. A second type consists of *variable flows*. Flows in this category can use larger amounts of bandwidth, and the bandwidths of the flows are linked in the sense that they will share available bandwidth proportionally. For example, three flows may have bandwidth requirements of 3, 4.5, and 9 Mbps relative to each other; the result of a corresponding Remos query may be that the flows will get 1, 1.5 and 3 Mbps respectively. A third type consists of *independent flows*, for which the user would like to know how much bandwidth is available after the requirements the first two classes have been satisfied. These can be viewed as lower priority flows.

The type of a flow is determined by the type of query made for the flow. Simultaneous flow queries may specify one set of flows of each type. This combination of different flow classes allows a wide variety of situations to be described concisely.

Simultaneous queries and sharing

Flows may share a physical link in the network. At bottleneck links, this means that flows are competing for the same resource, and each flow is likely to get only a fraction of the bandwidth that it requested. Of particular interest is the case where multiple flows belonging to the same application share a bottleneck link. Clearly, information on how much bandwidth is available for each flow in isolation is going to be overly optimistic. Remos resolves this problem by supporting queries for both individual flows, and simultaneously for a set of flows. The latter allows Remos to take resource sharing across application flows into account. Support for simultaneous flow queries is particularly important for parallel applications that use collective communication.

Determining how the throughput of a flow is affected by other messages being sent at the same time is very complicated and network specific. A variety of different sharing algorithms that affect the proportion of bandwidth received by a particular flow are deployed. While some networks have sharing policies that are precisely defined for certain types of traffic (e.g. ABR flows over ATM, or flows with bandwidth guarantees over FDDI II or ATM), on other networks (e.g. Ethernet), characterization of sharing behavior would require consideration of packet sizes, precise packet timings, queueing algorithms, and other factors. Moreover, how much bandwidth a flow gets depends on the behavior of the source, i.e. how aggressive is the source and how quickly does it back off in the presence of congestion.

It is unrealistic to expect Remos to characterize these interactions accurately in general. Our approach is to return the best knowledge available to the implementation that can be returned in a network-independent manner. In general Remos will assume that, all else being equal, the bottleneck link bandwidth will be shared equally by all flows (not being bottlenecked elsewhere). If other better information is available, Remos can use different sharing policies when estimating flow bandwidths. The basic sharing policy assumed by Remos corresponds to the max-min fair share policy [16], which is the basis of ATM flow control for ABR traffic [18, 2], and is also used in other environments [14].

4.3 Logical network topology

Remos supports queries about the network structure and topology in addition to queries about specific flows in the network. The reason we expose a network level view of connectivity is that certain types of questions are more easily or more efficiently answered based on topology information. For example, finding the pair of nodes with the highest bandwidth connectivity would be expensive if only flow-based queries were allowed.

Graphs are a well-accepted representation for network topology. Remos represents the network as a graph with each edge corresponding to a link between nodes; nodes can be either compute nodes or network nodes. Applications run only on compute nodes, and only compute nodes can send or receive messages. Network nodes are responsible only for forwarding messages along their path from source to destination. Each of the communication links is annotated with physical characteristics like bandwidth and latency.

Topology queries return the graph of compute and switch nodes in the network, as well as the *logical* interconnection topology. Use of a *logical topology graph* means that the graph presented to the user is intended only to represent how the network behaves as seen by the user, and the graph does not necessarily show the network's true physical topology. The motivation for using a logical topology is information hiding; it gives Remos the option of hiding network features that do not affect the application. For example, if the routing rules imply that a physical link will not be used, or can be used only up to a fraction of its capacity, then that information is reflected in the graph. Similarly, if two sets of hosts are connected by a complex network (e.g. the Internet), Remos can represent this network by a single link with appropriate characteristics.

In the absence of specific knowledge of sharing policies, we recommend that users of logical topology information assume that bandwidth is shared equally between flows. This assumption can be verified using queries. If other sharing policies become common, we could add a query type to Remos that would allow applications to identify the sharing policy for different physical links.

All information is represented in a network and system independent form. Hence, the network topology structure is completely independent of peculiarities of various types of networks and manages network heterogeneity in a natural way. Remos emphasizes information that can be collected on most networks, but it should be noted that not all types of information may be provided by all networks and Remos implementations. For example, if a network architecture provides specialized information like reliability or security of a link, Remos could be extended to include such information when available.

4.4 Data representation

Flow-based and topology queries can return both static (e.g. link capacity) and dynamic (e.g. link utilization) information. We discuss some of the issues associated with representing this information.

Variable timescales

Ideally, applications would like information about the level of service they will receive in the future. While some sophisticated networks may be able to actively predict performance, possibly based on input from applications on their future needs, most current applications will have to use past performance as an indicator of future performance. Remos supports queries about historical performance, as well as prediction of expected future performance. Initial implementations may only support historical performance, or use a simplistic model to predict future performance from current and historical data.

Different applications are also interested in activities on different timescales. A synchronous parallel application expects to transfer bursts of data in short periods of time, while a long running data intensive application may be interested in throughput over an extended period of time. For this reason, relevant queries in the Remos interface accept a timeframe parameter which allows the user to request data collected and averaged for a specific time window.

Statistical measures

Assuming a static environment, the performance of each network component can be approximated by a simple model representing bandwidth and message latency. The throughput of a flow will be the minimum of the bandwidth available on each of the links it traverses, taking the sharing policies into account. Estimating end-to-end latency is somewhat more complicated. Latency includes propagation delay, which is basically fixed given a route, plus queueing and forwarding delay in switches and routers. Estimating the latter requires a thorough network model that captures queueing effects, forwarding latencies, packet length, and scheduling effects. Such models have been developed, primarily in the context of developing support for guaranteed services. However, we adopt a simpler model that associates a fixed delay with network components. The reason is that precise characterization of delay is a fairly hopeless task given the presence of uncontrolled competing traffic sources. Moreover, we expect this simpler model to be sufficient for adaptive applications. Given this simple model, end-to-end latency can be obtained by simply adding up the delay introduced by each of the network components.

In reality, network information like bandwidth and latency will be variable because of variability introduced by competing users and inaccuracy introduced by the measurement techniques. As a result, characterizing these metrics by a single number would be misleading. For example, availability of a fixed small communication bandwidth represents a different scenario than the periodic availability of a high burst bandwidth, even though the average may be the same. Similarly, knowing that a certain bandwidth is definitely available represents a different scenario from a rough estimate of available bandwidth. To address these aspects, the Remos interface adds statistical information (variability and estimation accuracy measures) to all dynamic quantitative information. Statistical information allows applications to consider the range and variability of the information when they use it.

The most commonly used measure of variability is variance. Unfortunately, variance is only meaningful when applied to a normally distributed random variable. Network measurements, such as available bandwidth, are not necessarily normally distributed. In particular, the presence of bursty traffic would be expected to result in a bimodal or other asymmetric distribution of available bandwidth. Because this distribution is not necessarily known, and would be difficult to represent even if it could be determined by the network,

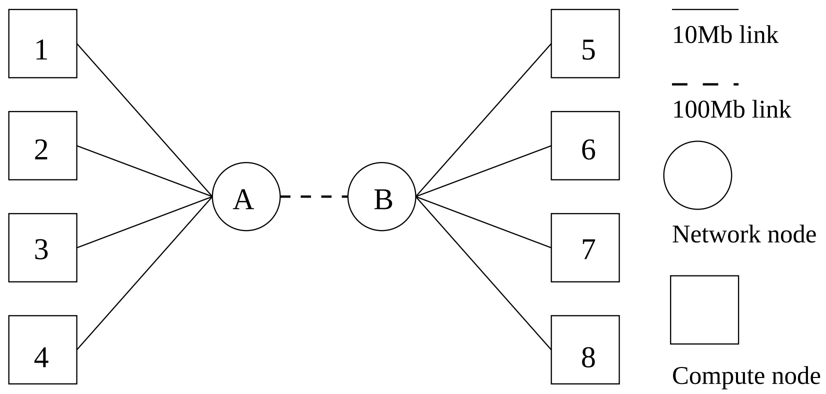

Figure 1: Remos graph representing the structure of a simple network. Nodes A and B are network nodes, and nodes 1–8 are compute nodes.

we present the variability of network parameters using quartiles, which is considered the best choice for an unknown data distribution [17].

4.5 Examples

Before we describe the actual Remos API, we would like to illustrate the abstraction of a "logical topology" using some simple examples.

Figure 1 shows a simple network represented by a graph. The links in this network representation are annotated with network performance information. However, it is just as important that the nodes include performance information as well. For instance, if nodes A and B each have an internal bandwidth of 100Mbps and all the compute nodes have bandwidths higher than 10Mbps, then the links connecting the compute nodes to the network nodes restrict bandwidth, and all node can send and receive messages at up to 10Mbps simultaneously. On the other hand, if nodes A and B have internal bandwidths of 10Mbps, then these two network nodes are the bottleneck and the *aggregate* bandwidth of nodes 1-4 and 5-8 will be limited to 10Mbps.

In the previous paragraph we (implicitly) assumed that Figure 1 represents a physical topology consisting of 8 workstations, 2 routers, and 9 links. However, Figure 1 can also be interpreted as a *logical* topology, potentially representing a broad set of (physical) networks. For example, if A and B have internal bandwidths of 10Mbps, it also represents two 10Mbps Ethernets, containing nodes 1–4 and 5–8 respectively, that are connected to each other with a 100Mbps link. While it may not correspond to the physical structure of the Ethernet wiring, it accurately represents its performance.

The importance of the distinction between compute and network nodes is illustrated in Figure 2. In Figure 2.a, a high speed "research" network has been added to compute nodes 1–4. Similar to the way most research networks are deployed, the nodes are dual homed, i.e., they are simultaneously connected to both the research and the "regular" network. Assuming that compute nodes cannot forward messages, node 1, for instance, cannot use the research network to reach any node other than nodes 2–4. If connectivity to other nodes is desired, one of the compute nodes can also serve as a router, as is logically shown in Figure 2.b. With this representation, node 1 can receive 10Mbps of data from node 5, and at the same time, send 10Mbps of data to node 6 through nodes C, D, A and B.

The sample network of Figure 2 also brings up the complications of routing. With simple networks such as the one shown in Figure 1, only one path connects every pair of nodes. However, in Figure 2.b, there are two possible routes between nodes 1–3 and the rest of the network. Because of the complexity of the routing issues, Remos does not export the routing information to applications, but may use this information to answer flow-based queries. Many networks have an acyclic topology, as shown in Figure 1. If multiple paths are possible, then only one path is likely to be of interest at a time, and other paths will not even show up in the logical topology. For example, most applications would only use the research network, or the regular network, but not both at the same time, so the Remos interface would only return one of the two

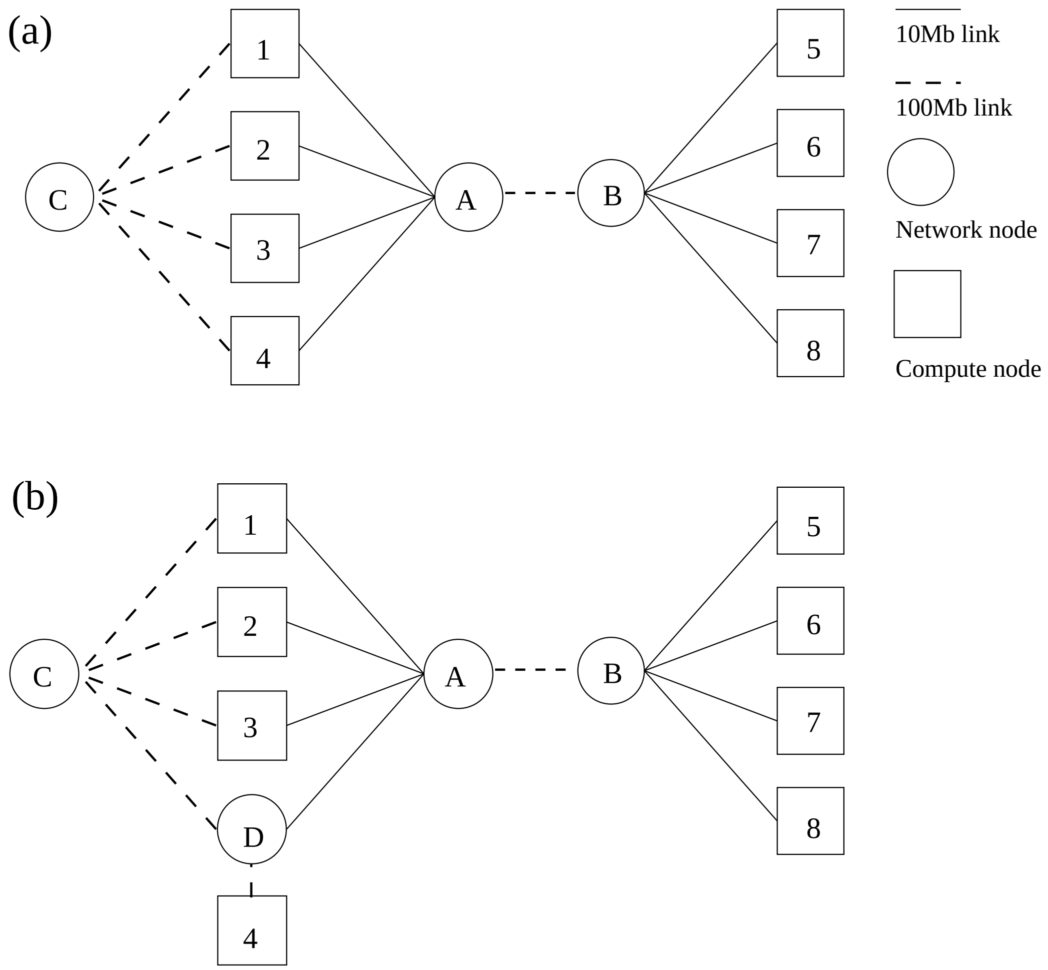

Figure 2: Network from Figure 1 with additional network added to nodes 1–4. (a) no routing between the two networks; (b) Node 4 has been modified to also route messages between the networks.

subnets, depending on the scenario. Remos may return a topology graph with multiple routes, when they are feasible, but no other routing information is returned. We expect that an application that makes use of such topology graphs is informed about how to properly interpret the information. Note that the results of both flow-based and topology queries implicitly include a lot of routing information.

4.6 Limitations

Our design of Remos ignores a number of important network properties. As we just mentioned, it deals with the issue of alternate routes in a restricted way. Moreover, it does not deal with multicasting, or with networks that provide guaranteed services. While both of these are important features that would be of interest to applications using Remos, the networks deployed today rarely make this functionality available to applications. As application-level multicast and guaranteed services become more widely available, we will extend Remos to support them.

5 Application programming interface

The Remos API is divided into three classes of functions: status functions, fitting functions, and topology queries. We briefly outline the main functions in this section. A more detailed description of the functions is given in Appendix A.

All queries support a timeframe field to specify the timeframe of interest to the application. Applications can specify queries that return information on previous, current, or a prediction of future network conditions. Physical characteristics can also be requested.

5.1 Status functions

Status functions return information on individual compute nodes and simple node-node flows. An additional function in this category allows the user to supply information on the software overhead contributed by the communication software to the Remos interface, so that such information can be taken into account when computing the responses to queries:

- **remos_get_node()** This provides information about a node's network characteristics.
- **remos_get_flow()** This provides information about the characteristics along a path between two nodes.
- **remos_node_delay()** This allows the user to contribute information about delays contributed by software layers unknown to Remos.
- **remos_get_node_info()** This call returns information about a node's compute power and load.
- **remos_node_query()** This call allows expandable queries to be made about a host. For example, currently a query can be made to obtain a list of shared libraries available on a machine, to ensure that an application can execute on it.

Note that the last two status functions in the above list provide information about compute nodes which is orthogonal to Remos goals, but they are included in the Remos interface for completeness.

5.2 Fitting functions

Fitting functions return information about the ability of a network to support several simultaneous flows. They implement the concept of flow based queries with sharing discussed in this document. Fitting functions allow an application to determine the service that will be received by a new set of flows, given the resource requirements of the existing flows and the sharing properties of the network (Section 4.2). The **remos_flow_info()** query allows the user to specify a set of end-to-end flows. The function then returns the bandwidth needs that can be met by the network. The **remos_flow_info()** query takes a set of parameters that can be used to describe a wide variety of scenarios.

- **fixed_flows** are flows that have specific bandwidth requirements that cannot be altered. If one cannot be met, the call will return failure, with an indication of what can be met. An important purpose of this flow class is that it allows the application to specify flows that the other two classes must compete against.
- **variable_flows** are used when an application is interested in determining how much bandwidth is available if it attempts to simultaneously send data in several flows. All flows in this class will be adjusted to rates that can be met by the network. The adjustment takes place in a approximately proportionate manner, based on the bandwidth requested for each flow and overall network constraints. A requested bandwidth is not reduced unless it is constrained by the network.
- **independent_flows** are used to determine how much bandwidth is remaining for additional flows, after meeting the first two needs. Each flow specified in the set is considered independently. An important use of this query is for deciding among several options for a planned communication.

This interface has the advantage that the user can obtain a proper answer for communication on networks with topologies and behaviors that may not be accurately representable through the topology interface.

A flag is provided to specify that a reservation should be placed for the resulting communication specification. The Remos interface does not support this reservation directly. However, when implemented on top of a network with QoS support, the reservation service would be a natural extension to Remos. A reservation request will be simply denied if such requests are not supported by the underlying network platform.

5.3 Topology functions

The third and final category is the topology query interface. This interface allows the user to obtain a network topology for a set of nodes selected by the user. As described in the design section, this topology is intended to represent the network's performance characteristics as seen by the application. Due to the variety of techniques for handling routing on networks, routing is not represented explicitly on the network. In most cases, a simple prediction based on the shortest path algorithm is appropriate, while in other cases, outside knowledge of the routing algorithm may be required.

6 Using the API

We outline how some sample applications can use the information retrieved by flow-based and topology queries to adapt to the network environment.

6.1 Adaptive applications

Some real-time applications, such as distributed video systems, require consistent quality of service to perform effectively. However, many applications with weaker real-time properties can adapt to the available network performance, for example by varying their frame rate, image quality, and processing load [8].

One system which can make use of such information is Odyssey[22], which manages a variety of adaptive applications across networks ranging from high-performance to wireless systems. Odyssey combines protocol specific adaptations, such as frame rate and quality, with function and data shipping, to select the best combination of network and computational resources to achieve the desired application quality metrics. Adaptation in Odyssey is based on a Viceroy, a controlling authority for the resources on each node. Each Viceroy interacts with Wardens that implement the adaptation mechanisms for specific distributed resource types. The Viceroy is currently responsible for collecting information on network status, but instead, it could use Remos to retrieve this information. In fact, a similar modification has already been made to support the status of wireless networks [9].

Video applications are interested in flows between a client node and a small set of server nodes. The Remos fitting function is the most appropriate in this case since it allows the application to ignore the rest of the network and focus only on the aspects of the network that affect the client's performance. The Viceroy can issue a flow-based query for each of the servers or server combinations, and select the servers that give the best throughput. It can periodically reissue the query, or it can ask for a callback when conditions in the network change considerably.

Adaptation is also widely used in the area of high performance computing. The Fx compiler [28, 27] has successfully used automatic analysis to assign nodes to meet the performance requirements of applications containing multiple tasks. However, this analysis assumes static network behavior, and a system like Remos is required for adaptation in a dynamic network environment. Two other examples of network-aware applications include a pipelined application that adapts the pipeline depth [23], and simple distributed matrix multiply that selects the optimal number of nodes [29]. In both cases, applications used simple benchmarks to characterizing network performance. Remos provides a simpler and possibly more accurate alternative.

6.2 Collective communication

Parallel message-passing applications are often very sensitive to communication performance. Of particular concern are collective operations, such as broadcast or vector reduction, where all nodes in the application are involved in communication simultaneously. Poorly chosen communication patterns for such operations result in bottlenecks and significant loss of performance [21]. Fortunately, implementations of MPI [10] can use the Remos interface to transparently optimize the communication needed by the application.

While the fitting interface could be used for planning collective communication, it would result in an excessive number of queries, as it is difficult to do anything other than a factorial set of queries to determine the appropriate pattern. The topology interface of the Remos system provides a more concise view of the network's structure. An MPI implementation can use the topology information to minimize communication on network bottlenecks. Once the set of candidate nodes has been narrowed down, the fitting interface could be used to get more accurate prediction for a small set of options, or the application could pick a set based on the logical topology only.

6.3 Clustering

A decision regarding what nodes to use for the execution of a parallel application can be made on the basis of availability, experience, performance, or efficient use of resources. Independent of the motivation, making these decisions requires some information on network and system performance. A typical problem is to locate the largest group of machines that have both a certain compute power and are capable of sustaining a certain bandwidth between themselves.

The network topology interface is the most appropriate mechanism to collect information to solve this problem. The interface can be used to connect candidate nodes with information on bandwidth availability on links connecting them. One of several known algorithms can then be used to deduce what nodes are "closest" together and will be able to meet connectivity requirements. Appendix C has a more detailed description of a simple greedy algorithm. Starting with an initial member of the group, the topology could be used to determine the bandwidth from the group to external nodes. At each step, the node with the highest incremental bandwidth is added to the group.

7 Implementation

We first discuss the options for collecting information needed by Remos and then outline our initial implementation.

7.1 Collecting information

The information needed by the Remos interface can be grouped into four categories: static topology information, dynamic bandwidth information, latencies, and future resource availability. We discuss each separately.

Static features of the network may have to be collected through a number of mechanisms. Information on routers and their connectivity can typically be collected through SNMP. Information about level 2 switching devices (e.g. bridges) may be harder to obtain in a systematic way, e.g., non-standard databases may have to be used. Some information, such as throughput limitations of devices, may also have to be collected by Remos in non-standard ways. One of the harder parts of collecting static information is to determine the scope of the network information needed. Application requests include a context, which is a list of hosts that are of interest. Remos has to determine, based on this list of hosts, what part of the network is relevant to the application's request. This requires using the routing tables to determine the paths a packet between a pair of listed hosts could take, and to identify all the network components used by these paths.

Dynamic bandwidth information is somewhat harder to track. In some cases, SNMP can again be used to retrieve that information from routers with appropriate Management Information Bases (MIBs). However, as in the case of static information, not all level 2 switching components provide this information. Similarly, Ethernets and other contention-based networks do not provide a simple way of retrieving the traffic load. It is possible to collect this information by setting up a node in promiscuous mode, i.e., it observes all traffic and can report the traffic load for that Ethernet segment, but this is an expensive solution. In general, collecting detailed bandwidth information for Ethernets or networks based on "dumb" switches, requires having each host report its network use and aggregating this information. Clearly this is not an attractive solution. Our hope is that, eventually, most switches will report dynamic traffic loads, and we can rely on standards methods such as SNMP to collect the bandwidth information.

Networks typically do not systematically keep track of *latency information*. As a result, the only way of collecting that information is through benchmarking. Note that benchmarking can be used to collect any type of information. For example, one could use a throughput benchmark to measure how much bandwidth is available between pairs of nodes. However, benchmarking is time consuming and introduces a significant amount of overhead in the network, so we use it only if there is no inexpensive option.

The estimation of *future network usage* is possibly the most complex task and requires the most sophisticated level of support. In a network that supports reservations, the reservation status can be used as a guide to future usage. The knowledge of usage patterns of executing applications, and estimates of how long they will execute, can also be a guide to future usage. However, in general it is a very challenging problem. We expect to be able to predict usage only in some scenarios in the course of this project.

7.2 Initial implementation

The initial implementation of the Remos interface is for a dedicated IP-based testbed. The testbed uses PCs running NetBSD as flexible routers, 100Mbps point-point Ethernet segments as links, and DEC Alpha systems as endpoints.

The Remos implementation has two main components, a Collector and Modeler, that are responsible for network-oriented and application-oriented functionality, respectively. The Collector consists of a process that retrieves information from routers using SNMP. This covers both static topology and dynamic bandwidth information. For latency, the Collector currently assumes a fixed per-hop delay, which is a reasonable approximation since this is a LAN testbed. A large environment may require multiple Collectors. The Collector is implemented in Java, since we envision that in the future it may be downloaded across the network. The Modeler is a library that can be linked with the application. It satisfies application requests based on the information provided by the Collector. Its primary tasks are generating a logical topology, associating appropriate static and dynamic information with each of the network components, and satisfying flow requests based on the logical topology.

This initial implementation is not specific to our testbed. It will run correctly on all networks that use routers that support SNMP. However, since it ignores some network components, such as layer 2 switches, it may generate inaccurate information on some networks. Once we have gained some experience with this version of Remos, we will extend it to include other network components. Support for confidence metrics will also be added

We plan to integrate Remos with the CMU Darwin system. Darwin [24] is developing sophisticated resources management techniques for networks. It combines hierarchical resource scheduling, support for virtual subnets, and customization of resource allocation and adaptation to create an "application-aware" network, i.e. a network that can be tailored to provide application-specific services. Customization of resource allocation and adaptation is achieved by downloading application code and state into the network, reflecting the use of of active networking [30]. Darwin can enhance Remos functionality in a variety of ways. For example, Darwin can provide application-specific feedback and implement application-specific callback conditions.

8 Related Work

A number of resource management systems have been developed that either allow applications to make queries about the availability of resources, or to directly manage the execution of the applications. Systems in present use primarily deal with computation resources, i.e., the availability and load on the compute nodes of the network. An application may use this information to control its own execution or the system may place the application on nodes that have the minimum load. Examples of resource managers include research systems like Condor [20] and commercial products like LSF (Load Sharing Facility). While such systems can be adequate for compute-intensive applications, they are not suitable for applications that handle movement of large data sets and applications based on internetworking, since they do not include a notion of communication resources.

More recently, resource management systems have been designed for internet-wide computing, some examples being Globus [11] and Legion [13]. These systems are large in scope and address a variety of mechanisms that are necessary to make large scale distributed computing possible. For example, Globus services include resource location and reservation, a communication interface, a unified resource information service, an authentication interface, and remote process creation mechanisms. The Remulac project and Remos interface are less ambitious, but more focused. We are concentrating on good abstractions for a network to export its knowledge to an application, and for applications to use this knowledge to achieve their performance goals. Thus, our research addresses some of the core problems that are critical to the development of large scale resource management systems, and we aim to develop a set of mechanisms and abstractions to allow the development of network-aware applications. Most network-aware applications require some policies to chose the appropriate mechanism. While these are beyond the scope of this project, they are the subject of other research efforts including the *Amaranth* project at Carnegie Mellon.

A number of groups have looked at the benefits of explicit feedback to simplify and speed up adaptation, e.g. [15, 9]. However, the interfaces used in these efforts have been designed specifically for the scenarios being studied.

A number of sites are collecting Internet traffic statistics, e.g. [1]. This information is not a in form that is usable for applications, and it is typically also at a coarser grain than applications would like to get. Another class of related research in the Internet is the collection and use of application specific performance data, e.g. a Web browser that collects information on what sites had good response times [25].

9 Concluding remarks

Remos allows network-aware applications to obtain information about their execution environment. Remos provides a uniform interface so that portable network-aware applications can be developed independent of any particular network architecture.

Network architectures differ significantly in the accuracy and depth of status information that can be obtained. They also differ in terms of how and where this information is collected and stored. Some legacy networks virtually provide no information at all, and Remos has to generate all information, possibly by performing application-level measurements (e.g., of latency). The benefit of a uniform interface is that Remos clients (application developers) are isolated from network details. When executing on a network architecture that provides timely and accurate information, such network-aware applications can immediately exploit the network's advanced features.

The challenges in defining the Remos interface are network heterogeneity, diversity in traffic requirements, variability of the information, and resource sharing in the network. Remos addresses these issues by offering a fairly high level of abstraction, thereby shielding the user from many of the details of the underlying network. Most applications should be able to use a flow-based query that provides information on the performance of application-level point-to-point flows. Applications have to specify all the flows they plan to use simultaneously so that Remos can account for the effects of sharing. For application that need lower-level information, a topology query primitive is provided. In this case, Remos returns a logical topology, i.e., a graph that only captures information of interest to the application.

One of the challenges in demonstrating the usefulness of Remos is to develop a portable and cost-effective implementation which has a low overhead. These features are essential for making Remos attractive to application developers. We are in the process of developing a prototype Remos implementation and expect it to validate the Remos design.

Appendices

A Remos API

We describe the data structures and functions used in the Remos interface.

A.1 Data structures

We have defined some common naming conventions for the types in this package:

Structure Names Structure types are named xxx_xxx_s.

Pointer Names For each structure type there is a corresponding pointer type named Xxx_Xxx. The remos routines nearly always pass around the pointer types rather than the structures.

List structures All remos list structures share a common layout, consisting of a maximum element count, a current element count, and an array of pointers (the elements). Users should use the provided routines for list management.

```
/* Error handling types */

/* Standard status values returned by Remos calls. */
typedef enum {
        REMOS_ERROR = -1,
        REMOS_OK = 0,
        REMOS_NA = 1 } Remos_Status;

typedef void*   Remos_Ptr;
typedef Remos_Error_s* Remos_Error; /*opaque type for error info */
typedef void    (*Remos_Handler)(Remos_Error* error, Remos_Ptr errarg);

/* Abstract remos_list and list_element.  The template for all of the
   Remos_Xxx_List types.
*/

typedef void* Remos_List_Element
typedef struct{
  int elem_max;
  int elem_count;
  Remos_List_Element list[];
} remos_list_s;
typedef remos_list_s* Remos_List;


/* Data structure for a flow.
    A flow represents a unidirectional flow of data from a source to a
    destination node.  The latency reported is the sum of the latencies
    along each step on the path, and the bandwidth is the bottleneck
    bandwidth along the path.  These numbers include contributions from
    the source and destination nodes.  */
typedef struct{
  remos_remulac_id  src_node;
  remos_remulac_id  dst_node;
```

```
  double latency;
  double bandwidth;

  remos_stat *latency_stats;
  remos_stat *bandwidth_stats;

} remos_flow_s;

typedef remos_flow_s* Remos_Flow;

/* Data structure for a list of flows.  Packages up the array and
   element count together.
*/
typedef struct{
  int elem_max;
  int elem_count;
  Remos_Flow list[];
} remos_flow_list;
typedef remos_flow_list* Remos_Flow_List;


/* A general data structure for representing the variability in network
   performance data. A maximum, minimum, and average are given. Variance
   is reported using quartiles, since the underlying distribution is almost
   certainly not gaussian.  A measure of confidence in the results is also
   reported, although the current implementation does not actually
   support this.
*/
typedef struct{
  double max;
  double min;
  double expected;

  /* index of dispersion */
   double q1, q3;

   int   confidence;
} remos_stat_s;

typedef remos_stat_s* Remos_Stat;

/* Data structure that represents a computation or a communication
   (switch) node.  Bandwidth and latency information is provided for each
   node.  In the switch case it indicates the internal switch fabric.  In
   the compute node case, it indicates the costs associated with processing
   a message and moving it to the network interface.
*/
typedef struct{
  char *node_name;
  remos_remulac_id id;
  int ip_address;
  int node_type;                /* compute or switching */

  double latency;
```

```
  double bandwidth;

  Remos_Stat latency_stats;
  Remos_Stat bandwidth_stats;


} remos_node_s;


/* Data structure for a list of nodes.  Packages up an array and a
   max_element together
*/
typedef struct{
  int max_elem;
  int elem_count;
  Remos_Node list[];  /* array of pointers */
} remos_node_list_s;
typedef remos_node_list_s* Remos_Node_List;



/*************
  Begin graph structures
  ************/
typedef struct{
  int max_elem;
  int elem_count;
  Remos_Link list[];  /* array of pointers */
} remos_link_list_s;
typedef remos_link_list_s* Remos_Link_List;


/* Abstract topology graph for a set of nodes. It contains a list
   of flows and a list of nodes  */
typedef struct{
   Remos_Link_List  edges;
   Remos_Graph_Node_List nodes;
} remos_graph_s;
typedef remos_graph_s* Remos_Graph;


/* this structure for representing the links in a graph is identical to
   the flow record, with two exceptions:
      - it does not include the communication characteristics of the
          endpoints
      - it uses pointers to refer to nodes which are guaranteed to be
          unique in the graph structure
*/
typedef struct{
  remos_graph_node_s *src_node;
  remos_graph_node_s *dst_node;

  double latency;
  double bandwidth;
```

```
  Remos_Stat latency_stats;
  Remos_Stat bandwidth_stats;
} remos_link_s;
typedef remos_link_s* Remos_Link;


typedef struct{
   Remos_Link link;
   Remos_Graph_Node node;
} remos_neighbor_s;
typedef remos_neighbor_s* Remos_Neighbor;

typedef struct{
  int max_elem;
  int elem_count;
  Remos_Neighbor list[];  /* note: list of ptrs! should it be list-of-struct?*/
} remos_neighbor_list_s;
typedef remos_neighbor_list_s* Remos_Neighbor_List;

/* this is the structure for representing a node in a graph */
typedef struct{
  char *node_name;
  remos_remulac_id id;
  int ip_address;
  int node_type;                /* compute or switching */

  Remos_Neighbor_List  neighbors;

  double latency;
  double bandwidth;

  Remos_Stat latency_stats;
  Remos_Stat bandwidth_stats;
} remos_graph_node_s;
typedef remos_graph_node_s* Remos_Graph_Node;


/* this is identical to the link_record, but included remulac
   ids.  It is returned only by the raw_graph interface,  which is
   used by the remos_get_graph function.  Normally a user does not
   need to use this structure.  */
typedef struct{
  remos_remulac_id  src_node;
  remos_remulac_id  dst_node;

  double latency;
  double bandwidth;

  Remos_Stat latency_stats;
  Remos_Stat bandwidth_stats;
} remos_raw_link_record;
typedef remos_raw_link_s* Remos_Raw_Link;
```

A.2 Function interface

A.2.1 Utility functions

remos_start

An application uses `remos_start` to initialize the Remulac system. This function may include opening a connection with the lower layer, initializing memory structures, etc. It accepts the application name as an argument. This name should be unique across the system. The name is intended for use by this and lower layers in identifying sets of traffic flows. The name is for convenience of the system only. There is no requirement that all nodes in an application make this call, and no additional information will be available because other nodes have called this function. Note, however, that if all nodes do not call this function, it may be more difficult for previous communications done by an application to be excluded from future predictions if that same application makes further calls after performing communication.

This call returns the Remulac ID of the calling node in the argument `callers_id`. This ID is unique on each processor in the system and is used for identifying nodes. Both compute and switching nodes share the same set of Remulac IDs.

The caller may also use this function to establish a default error handling routine and error communication data block. The return value of the routine will be one of the standard `Remos_Status` values.

```
Remos_Status remos_start (char* name,
                          void* errarg,
                          remos_remulac_id* callers_id,
                          Remos_Handler errhand,
                          Remos_Error* error);

        name:   name for application --- should be shared only by other
                processes of the same task
        callers_id: address of a remos_remulac_id.  Will be filled in
                with the Remulac ID of the calling node.  Must be present.
        errarg: pointer to an application defined block of data used
                to communicate error information from the shared error
                handler routine to the rest of the application.  May
                be NULL.
        errhand: address of the application default error handler
                routine. May be NULL.
        error:  address of a Remos_Error descriptor for error
                reporting. May be NULL.
```

remos_finish

`remos_finish` is used to indicate the application is finished using the interface. It does not imply that the application itself is exiting. It is intended so any resources allocated by the interface internally can be released. `remos_finish` takes a standard error argument, and returns a standard `Remos_Status`.

```
Remos_Status remos_finish(Remos_Error* error);
```

remos_errno

`remos_errno` is used to extract more specific error information from a `Remos_Error`. It will return one of the (as yet undefined) specific remos error codes. Most remos routines return a `Remos_Status` to indicate success or failure. `remos_errno` is the main way for users to detect which error occurred.

```
int remos_errno(Remos_Error error);
        error:  a remos_error returned from one of the other remos
                routines.
        return: the error number specified by error
```

remos_errmsg

`remos_errmsg` returns a pointer to a null-terminated error message string corresponding to the error specified by `error`. This string should not be deallocated by the user.

```
const char* remos_errmsg(Remos_Error error);
```

remos_myid and remos_getid

These functions are used to obtain the remulac ID of the calling node or of a remote node. Both routines return a standard `Remos_Status` value to indicate success or error.

```
Remos_Status remos_myid(remos_remulac_id* callers_id,
                        Remos_Error* error);
        callers_id: address of a remos_remulac_id to be filled in with
                the Remulac ID of the calling node.  May not be NULL.
                Value is unchanged on error.
        error:  Address of a Remos_Error structure for error
                reporting.  May be NULL.


Remos_Status remos_getid(int ip, int p,
                         remos_remulac_id* result_id,
                         Remos_Error* error);
        ip:     ip address of node
        p:      processor number (default of 0)
        result_id: the requested id.  May be null if an error occurs.
        error:  Address of a Remos_Error structure for error
                reporting.  May be NULL.
```

remos_hostlist

To determine which nodes Remulac is aware of, the `remos_hostlist` can be called. It accepts an optional mask to restrict the range queried and returns a list of nodes. It allocates a `Remos_Hostlist` structure which the list of nodes and the number of nodes in the list. It returns a standard `Remos_Status`.

```
Remos_Status remos_hostlist(int mask,
                            Remos_Hostlist* hostlist,
                            Remos_Error* error);
        mask:   a subnet mask expressed as an int
        hostlist: the address of a Remos_Hostlist pointer.
                remos_hostlist will allocate a Remos_Hostlist and fill
                it in with the list of known hosts, returning the
                pointer here.
        error:  the address of a Remos_Error record for error
                reporting.  May be NULL.
```

A.2.2 List Functions

These functions provide common support for operations on remos lists.

remos_new_list

```
Remos_List remos_new_list(int elem_max);
        elem_max: The number of elements this list may hold without
                reallocation.  May be 0.
        return: a pointer to an allocated and initialized
                Remos_List.  The caller should cast this to the
                desired Remos_Xxx_List type.
```

remos_append

This function supports appending a list_element to a `remos_list`. It is actually implemented as a macro, so callers need not cast arguments. `to_list` will be updated appropriately; the internal list array will be realloc-ed if necessary.

```
void  remos_append(Remos_List_Element the_element,
                   Remos_List to_list)
        the_element: The element to be appended to the logical end of
                the list
        to_list: the list to be appended.
```

remos_list_all_elements

This function supports iteration over the elements of a `Remos_List` without access to internal list structures. It is actually a macro suitable for use as the entire loop control of a `for` statement. Callers need not type-cast the arguments, but should be careful to ensure that the provided list pointer and element pointer have compatible types—the macro makes no attempt to check this for you! The example in Appendix C makes use of this macro.

```
remos_list_all_elements(Remos_List list,
                        Remos_Elem elem_iter, int i)
        list:      The list to iterate over
        elem_iter: a pointer to the element type of list.  It will be
                iterated over all members of the list.  It is the
                caller's responsibility to make sure the elem_iter has
                the same type as the list element.  The iterator WILL
                NOT CHECK!
        i:         supplied for internal use by the macro
```

A.2.3 Status functions

remos_get_node

`remos_get_node` obtains information about a node. It returns a standard `Remos_Status`.

```
Remos_Status remos_get_node(
                remos_remulac_id node,
                remos_node_record **data,
                double *timeframe,
                Remos_Error* error);

        node:   the Remulac ID of the node queried for
        data:   returns pointer to an allocated structure for the information
        timeframe: frame of reference for this query
        error:  the address of a Remos_Error record for error
                reporting.  May be NULL.
```

remos_get_flow

This function obtains information about a path in the network. It accepts two arguments for the start and end node of the path and fills in the structure with information about this path. A status is returned. This routine is syntactic sugar for the `remos_flow_info` call, discussed below.

```
Remos_Status remos_get_flow(remos_remulac_id source,
                            remos_remulac_id destination,
                            Remos_Flow *data,
                            double *timeframe,
                            Remos_Error* error);
        source: compute node at which flow originates
        destination: compute node at end of flow
        data:   (out) returns a pointer to allocated structure with this
                information
        timeframe: frame of reference for this query
```

remos_node_delay

In order to capture the effects of higher-level systems, such as RPC and message passing, on communication performance, a call is provided which can inform the network layer about latency and bandwidth restrictions imposed by layers which the network subsystem is not aware of. This function takes a list of nodes and stores the latency and bandwidth restrictions for each node internally, for use with future queries. Only one entry may be stored in this manner, so if there are delays caused by multiple layers, these must be combined before being given to the system.

```
remos_node_delay(remos_node_list *nodes, int n, double timeframe);
```

A.2.4 Fitting functions

remos_flow_info

The purpose of this function is to determine whether the network can satisfy a particular set of communication requirements. This is a "best-effort" function and the accuracy of its results will vary widely depending on the capabilities of the underlying network. It returns a standard `Remos_Status`.

The first field is a list of fixed-rate flows. These are used for data which must be sent at a given rate. If this set of requirements cannot be satisfied, the function will return `REMOS_ERROR`. If these requirements are present and can be met, it returns `REMOS_OK`.

The second field is a list of flows which can be adjusted. If the requested rates cannot be achieved on all of these flows, some or all of the flow rates will be reduced, and the modified results will be returned. The reductions are intended to be proportional to the demands, although the exact policy is implementation and network specific.

The third field can be used to determine how much traffic could be supported on another flow after the first two sets of requirements have been met. A flow in this field will be filled with the maximum rate which could be supported considering the other flows in the first two fields. A set of flows can be listed here, but the rate of each flow will be calculated independently. Thus, a set of flows listed here could be used to determine which flow to use to transfer a single set of data, because there are no contentions to be considered. However, there is no guarantee that all capacities can be met simultaneously, so a simultaneous pair of transfers could not make use of this field.

Any of the three lists of flows can be empty or NULL. If all three lists are empty, `remos_flow_info` will return `REMOS_NA`.

The flags may include a RESERVE flag, in which case the result returned will be reserved by the network. `Remos_flow_info` will return `REMOS_NA` if reservations are not supported.

Note that the bandwidths reported by this query are not enforced by the system. If communication is intended to function as indicated in the results of a query, it is up to the application or a communication layer above the Remulac system to restrict the bandwidth of each node to the level which can be supported.

Both bandwidth and latency are considered by this call. However, it is usually not possible to request lower latencies, so a low latency requirement will frequently result in an unsatisfiable request. The latency expected on the flows will be filled in when latency set to zero on the query, otherwise it will return failure if the requested latency cannot be met.

```
Remos_Status remos_flow_info(Remos_Flow_List fixed_flows,
                             Remos_Flow_List variable_flows,
                             Remos_Flow_List flow_queries,
                             int flags, double* timeframe,
                             Remos_Error* error);
        fixed_flows:    list of flows which must be met.  May be length 0
                        or NULL.
        variable_flows: flows which can be reduced proportionately.
                        May be length 0 or NULL.
        flow_queries:   list of flows to be filled in with capacity after
                        meeting the fixed_flows and variable_flows
                        requirements.  May be length 0 or NULL.
        flags:          flags for options, currently RESERVED is only flag
                        otherwise 0
        timeframe:      frame of reference for this query
        error:          Address of a Remos_Error structure for error
                        reporting.  May be NULL.
```

A.2.5 Graph functions

remos_get_graph

`remos_get_graph` is used to obtain a graph structure for a set of nodes. A list of nodes is passed to it. All structures pointed to by the returned graph will have been allocated just for this purpose and no external structures will be linked, including the nodes pointed to in `nodes`.

```
Remos_Status remos_get_graph(Remos_Node_List nodes,
                             Remos_Graph* data,
                             double *timeframe,
                             Remos_Error* error);
        nodes: list of compute nodes to include in graph
        data: pointer to allocated graph structure
        timeframe: frame of reference for this query
        error:          Address of a Remos_Error structure for error
                        reporting.  May be NULL.
```

remos_raw_graph

`remos_get_graph` is really just a wrapper function to `remos_raw_graph`, which is the function which actually spans the interface. This query accepts a list of compute nodes and returns a list of links and compute and switch nodes necessary to form a logical graph. This is a rather ugly format and most users will feel more comfortable calling the utility function `remos_get_graph`.

The array of nodes which is passed to `remos_get_graph` will have the set of switching nodes required to form the virtual topology appended to it.

This function must always return a connected graph. The list of links which is returned forms the set of edges in the graph and the complete list of nodes, with switching nodes appended, forms the vertices. `n_nodes` must be set the number of nodes upon entry and will be modified to include the number of additional switching nodes added. `n_links` will be set upon return to the number of links in the graph. The `remos_get_graph` function is normally used to convert this into a more usable format with adjacency lists.

```
Remos_Status remos_raw_graph(Remos_Node_List *nodes,
                             Remos_Rawlink_List *links,
                             double timeframe,
                             Remos_Error* error);

        nodes:  Address of the list of nodes of interest.  If
                additional switching nodes are needed, they will be
                appended to the list.
        links:  Address of an (empty) list of links.  The links between
                the nodes will be appended to the list.
        timeframe: Timeframe for this graph in seconds.  Negative
                values indicate request for history; positive request
                prediction of the future.
        error:  the address of a Remos_Error record for error
                reporting.  May be NULL.
```

B System API

The system interface complements the network interface by providing information on the internal details of the compute nodes. It is not part of Remos, but the information it provides is essential to make use of a distributed system.

B.1 Overview

1. `cmclsys_get_node_info()` This call returns the dynamically adjusted info about a node.

2. `cmclsys_node_query()` This call allows expandable queries to be made about a host.

B.2 Data Structures

```
/* all time lengths which are stored in doubles are in units of seconds */

typedef struct {
  remos_remulac_id node;      /* remulac node id */
  int p;                      /* number of processors */


  char* node_name;            /* DNS name of node */
  int ArchID;                 /* each unique (incompatible) arch has a
                                 unique number */
  char* vendor;               /* vendor of machine ("DEC", "DELL")*/
  char* cpu;                  /* CPU type ("ALPHA","Pentium II","x86") */
  char* os;                   /* OS ("DUX","Linux","NT") */
  char* version;              /* vendor dependent format ("4.0c","2.0.12",
                                 "4.0WS") */

  struct in_addr NetworkID;   /* IP address */
  int SharedMemClass;         /* shared memory category (preliminary
                                 implementation is to assign IDs to types,
                                 probably need much more thorough support) */

  /*  this is the time frame for which these results are reported.  The
    interface will attempt to report the closest available time interval
    to the one which was requested but may be forced to convert from 3
```

```
    minutes into past to 5 minutes into past if that's what's available.
    This will be negative for values in the past.   Units are seconds. */
  double TimeFrame;

  /* these are adjusted based on the time-frame of the record */
  int Processors;          /* number of usable processors */
  double PhysicalMemory;   /* available RAM (bytes) */
  double DiskSpace;        /* usable local filesystem storage (bytes)*/
  double VirtualMemory;    /* available local virtual memory (bytes) */
  int Users;               /* current # of interactive logins */
  double LoadAverage;      /* load average (number of runnable jobs) */

  /* these times are not adjusted to reflect the TimeFrame variable */
  double Benchmark;              /* performance rating (of fully available
                                    processor) */
  double MachineUp;              /* time machine has been up (s) */
  double DaemonUp;               /* time daemon has been up (s) */
  double ConsoleUse;             /* console idle time (s) */
  time_t ObservationTime;        /* time of last observation */

} cmclsys_node_info_s;

typedef cmclsys_node_info_s* Cmcl_Node_Info;
```

B.3 Function Interface

The basic function `cmclsys_get_node_info()` accepts an argument giving the node for which the information is needed and an argument which indicates the time frame for which the results are to be reported. Zero indicates current observations, positive numbers request a prediction for that many seconds into the future, and negative numbers request data for that many seconds of history. Physical information may be requested by passing `CMCLSYS_PHYSICAL` as the timeframe. For the physical information query, the `TimeFrame` in the returned structure will be set to `CMCLSYS_PHYSICAL`, and the uptime and related fields will reflect the current status. The function passes a pointer to a `Cmcl_Node_Info` structure, which will be filled in with the information. The `TimeFrame` field in the retured structure will indicate the actual time period used to report the data, which may vary somewhat from the requested period.

```
Remos_Status cmclsys_get_node_info(remos_remulac_id id, Cmcl_Node_Info info,
                                   double timerequest, Remos_Error* error);
            id: remulac id of node information is requested for
          info: pointer to structure info to be put into
   timerequest: timeframe for which info is desired
         error: Address of a Remos_Error structure for error
                reporting.  May be NULL.
```

The function `cmclsys_node_query()` is designed to allow queries which were not anticipated in this interface or which do not easily fit in the `Cmcl_Node_Info` structure. If the application and node are aware of a query type unknown to this interface, they can use this function.

The well-known query name is passed in `queryname`. A pointer to a user-allocated `buffer` which is to be used for in/out communication is also required. The allocated length of the buffer is passed and the number of bytes of the buffer used for output is returned. A zero length is returned if the query is unknown, otherwise a negative length is returned indicating the desired buffer length, in the case that the buffer was too small. We suggest that any queries which are added using this interface develop their own protocol for returning errors through the buffer.

```
Remos_Status cmclsys_node_query(remos_remulac_id id,
                                char* queryname, char* buffer,
                                int* buflen, Remos_Error* error);
           id: remulac id of node information is requested for
    queryname: well-known name for information
       buffer: used for passing data in and out
       buflen: on call, indicates length of buffer, on return,
               indicates length of data returned, 0 for query unknown,
               and negative numbers indicating the required length of
               buffer
        error: address of a Remos_Error descriptor for error
               reporting.  May be NULL.
```

To facilitate applications which can require certain optional software support to run, a query can be made to determine whether certain shared libraries (otherwise known as DLLs) are available on a platform. To query if a list of libraries are available on a particular machine, submit a query with a queryname of "SharedLibraryQuery" and a newline separated list of libraries to be searched for. A `NULL` indicates the end of the list. If a specific version is required that is not included in the library name, that version number should be separated from the library name with a comma. Upon return, `buffer` will be replaced with a newline separated list of "yes" and "no" fields. If a specific version was requested and not available, but another version of a library was available, the "no" will be followed by a comma separated list of available versions. A blank query will return a list of all libraries available on that node.

C Clustering example

The following code demonstrates how the Remos API and data structures are used to build a cluster of tightly coupled machines using a greedy heuristic. This algorithm is selected for clarity of presentation rather than as an ideal way of solving this problem.

The cluster is chosen as a set of n processors that are *close* to a designated node. The criterion for closeness is the average time to send a message of a given size to other nodes in the cluster. The input is a list of nodes, a designated start node, the size of the desired cluster, and a message size. The return value is a cluster of selected nodes. At each step, the routine looks at the nodes not in the cluster and adds the one which has the lowest average communication time with nodes already in the cluster. The communication times between nodes in the cluster and those not in it are calculated using `compute_msgtimes`.

A `NetworkParam` data type is assumed to exist which can store destination node, latency, bandwidth, and communication time for that node from the origin node. A `NetworkCost` structure is assumed which can store a set of `NetworkParam` data types, indexed by node.

```
/* make_cluster: routine to use greedy algorithm to form a cluster
 *    start_host: initial node to begin forming cluster around
 *             n: number of nodes to be in cluster
 *         nodes: all candidates for the cluster
 *        length: size of message for determining cost
 *
 *       returns: list of nodes in cluster
 */
Remos_Node_List make_cluster ( Remos_Node start_host,
                             int n, Remos_Node_List nodes, int length){
   NetworkCost pathCosts[n];

   RemosGraph graph;
   double timeframe = 0;
```

```
   Remos_Node_List hosts; /*actual members*/

   /* get graph containing all nodes of interest */
   remos_get_graph(nodes, &graph, &timeframe, NULL);

   /* create a list to store members of cluster and add the starting
      node to that list */
   hosts = remos_new_list(n);
   remos_append(start_host, hosts);

   /* calculate times to send a message from the starting node to
      all others */
   NetworkCost[0] = compute_msgtimes(graph, start_host, length);
   cluster_size=1;

   while(cluster_size<cluster_n){

     Remos_Node new_host;

     /* pick the node which has the minimum average communication time
        from all nodes already in the cluster */
     new_host = pick_minimum_average_distance(NetworkCost, nodes,
                                              cluster_size);
     /* add to cluster list */
     remos_append(new_host, hosts);

     /* calculate time from the new node to all other nodes */
     NetworkCost[cluster_size] = compute_msgtimes(graph, new_host, length);

     cluster_size++;
  }

  return hosts;
}
```

The following routine, `compute_msgtimes()`, determines the time it will take to send a message of fixed size from one node to all other nodes in the network. The input parameters are the graph, the start node, and the message length. The output is a `NetworkCost` structure with `NetworkParam` members indicating the time to send a message to each node in the graph. The function `compute_distances` is used to find the latency and bandwidth information.

```
/* compute_msgtimes: computes the time it takes to send a fixed
 *                      length message to a set of nodes
 *    all_nodes: list of all nodes involved (source and destinations)
 *         node: index of source in all_nodes
 *   msg_length: length of message of interest
 *
 *      returns: array of times to send to each node in all_nodes
 */
NetworkCost compute_msgtimes(Remos_Graph graph, Remos_Node node,
                              int msg_length){
  NetworkCost pathCosts = newNetworkCost(graph->nodes->elem_count);
  NetworkParam thisPath;

  compute_distances(node, pathCosts);
```

```
  for(all_elements(pathCosts, thisPath, i))
    if(thisPath->node != node)
      put_param(pathCosts,thisPath.latency, thisPath.bandwidth,
                thisPath.latency+msg_length/thisPath.bandwidth);

  return pathCosts;
}
```

The following procedure, `compute_distances()`, finds the latency and bandwidth between a designated node and each of the other nodes in the network. Latency is the sum of latencies along a path and bandwidth is the minimum of the bandwidths of the links on the path.

```
/* compute distances: return distance from one node to all others
 *       node: pointer to current node in traversal
 *  pathCosts: NetworkCost structure which adds NetworkParam's as
 *             nodes are encountered in the traversal
 */

 void compute_distances(Remos_Graph_Node node, NetworkCost pathCosts){
   NetworkParam toGetHere = get_param(pathCosts, node);

   for(remos_list_all_elements(node->neighbors, neighbor_i, i)){
     NetworkParam info = get_param(pathCosts, neighbor_i);
     if(info == UNVISITED){
       put_param(pathCosts,
                 neighbor_i, neighbor_i->node->latency+toGetHere->latency,
                 min3(neighbor_i->node->bandwidth,toGetHere->bandwidth
                      neighbor_i->link->bandwidth),0 /* time for message*/);
       compute_distances(neighbor_i.node);
     }
   }
 }
```

References

[1] http://www.lanr.net/INFO.

[2] ATM User-Network Interface Specification. Version 4.0, 1996. ATM Forum document.

[3] Bao, H., Bielak, J., Ghattas, O., O'Hallaron, D. R., Kallivokas, L. F., Shewchuk, J. R., and Xu, J. Earthquake ground motion modeling on parallel computers. In *Proceedings of Supercomputing '96* (Pittsburgh, PA, Nov. 1996).

[4] Bolliger, J., and Gross, T. A framework-based approach to the development of network-aware applications. In preparation.

[5] Bonomi, F., and Fendick, K. The rate-based flow control framework for the available bit rate atm service. *IEEE Network Magazine 9*, 2 (March/April 1995), 25–39.

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

[7] Chandra, P., Fisher, A., Kosak, C., and Steenkiste, P. Experimental evaluation of atm flow control schemes. In *IEEE INFOCOM'97* (Kobe, Japan, April 1996), IEEE, pp. 1326–1334.

[8] Clark, D., Shenker, S., and Zhang, L. Supporting real-time applications in an integrated services packet network: Architecture and mechanisms. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols* (Baltimore, August 1992), ACM, pp. 14–26.

[9] Eckhardt, D., and Steenkiste, P. A Wireless MAC with Service Guarantees. In preparation, 1998.

[10] Forum, T. M. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93* (Oregon, November 1993), ACM/IEEE, pp. 878–883.

[11] Foster, I., and Kesselman, K. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications*. To appear.

[12] Geist, G. A., and Sunderam, V. S. The PVM System: Supercomputer Level Concurrent Computation on a Heterogeneous Network of Workstations. In *Proceedings of the Sixth Distributed Memory Computing Conference* (April 1991), IEEE, pp. 258–261.

[13] Grimshaw, A., Wulf, W., and Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM 40*, 1 (January 1997).

[14] Hahne, E. L. Round-robin scheduling for max-min fairness in data networks. *IEEE Journal on Selected Areas in Communication 9*, 7 (September 1991).

[15] Inouye, J., Cen, S., Pu, C., and Walpole, J. System support for mobile multimedia applications. In *Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video* (St. Louis, May 1997), pp. 143–154.

[16] Jaffe, J. M. Bottleneck flow control. *IEEE Transactions on Communications 29*, 7 (July 1981), 954–962.

[17] Jain, R. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

[18] Jain, R. Congestion control and traffic management in ATM networks: Recent advances and a survey. *Computer Networks and ISDN Systems* (February 1995).

[19] Kung, H., Blackwell, T., and Chapman, A. Credit update protocol for flow-controlled ATM networks: Statistical mulitplexing and adaptive credit allocation. In *Proceedings of the SIGCOMM '94 Symposium on Communications Architectures and Protocols* (August 1994), ACM, pp. 101–114.

[20] Litzkow, M., Livny, M., and Mutka, M. Condor — A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems* (San Jose, California, June 1988).

[21] Lowekamp, B. B., and Beguelin, A. Eco: Efficient collective operations for communication on heterogeneous networks. In *Proceedings of the 10th International Parallel Processing Symposium* (April 1996), pp. 399–405.

[22] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J., Flinn, J., and Walker, K. Agile application-aware adaptation for mobility. In *Proceedings of the Sixteenth Symposium on Operating System Principles* (October 1997), pp. 276–287.

[23] Siegell, B., and Steenkiste, P. Automatic selection of load balancing parameters using compile-time and run-time information. *Concurrency - Practice and Experience 9*, 3 (1996), 275–317.

[24] Steenkiste, P., Fisher, A., and Zhang, H. Darwin: Resource management in application-aware networks. Tech. Rep. CMU-CS-97-195, Carnegie Mellon University, December 1997.

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

[26] Subhlok, J., Steenkiste, P., Stichnoth, J., and Lieu, P. Airshed pollution modeling: A case study in application development in an HPF environment. In *12th International Parallel Processing Symposium* (Orlando, FL, April 1998). To appear.

[27] Subhlok, J., and Vondran, G. Optimal latency–throughput tradeoffs for data parallel pipelines. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures* (Padua, Italy, June 1996), pp. 62–71.

[28] Subhlok, J., and Yang, B. A new model for integrated nested task and data parallel programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June 1997), ACM.

[29] Tangmunarunkit, H., and Steenkiste, P. Network-aware distributed computing: A case study. In *Second Workshop on Runtime Systems for Parallel Programming (RTSPP)* (Orlando, March 1998), IEEE, p. Proceedings to be published by Springer. Held in conjunction with IPPS '98.

[30] Tennenhouse, D., and Wetherall, D. Towards and active network architecture. *Computer Communication Review 26*, 2 (August 1995), 5–18.