A Resource Query Interface for Network-Aware Applications

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

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

Networked systems provide a cost-effective platform for parallel computing, but the applications have to deal with the changing availability of computation and communication resources. *Network-awareness* is a recent attempt to bridge the gap between the realities of networks and the demands of applications. Network-aware applications obtain information about their execution environment and dynamically adapt to enhance their performance. Adaptation is especially important for synchronous parallel applications since a single busy communication link can become the bottleneck and degrade overall performance dramatically. This paper presents Remos, a uniform API that allows applications to obtain relevant network information, and reports on the development of parallel applications in this environment. The challenges in defining a uniform interface include network heterogeneity, diversity and variability in network traffic, and resource sharing in the network and even inside an application. The first implementation of the Remos system is hosted on an IP-based network testbed. The paper reports on our methodology for developing adaptive parallel applications for high-speed networks with Remos, and presents results that highlight the importance and effectiveness of adaptive parallel computing.

Index terms: Parallel network computing, parallel programming, network interface, performance measurement, resource discovery

Effort sponsored by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287.

*Jaspal Subhlok is currently with Dept. of Computer Scince, University of Houston, Houston, TX 77098

1 Introduction

Clusters of networked computers are an attractive and ubiquitous platform for a wide range of parallel and distributed applications. However, effective use of networked resources is still a major challenge. A networked system consists of compute nodes (hosts), network nodes (routers and switches), and communication links. Network conditions change continuously due to the sharing of resources, and when resource demands exceed resource availability, application performance suffers. Computation load as well as congestion on network nodes and links can reduce the application response time dramatically. An attractive way of dealing with such changes is to make applications *network-aware*, i.e., the application periodically adapts to the network execution environment in an application-specific manner. E.g., an application may select the optimal set of nodes for execution based on compile time information about the kind of communication steps that are included in the program and based on runtime information about the system state. An application may periodically check the resource availability of other nodes and decide to migrate the computation. Or an application may want to select one of several links to transfer data, based on dynamic properties of the links.

The Remos system developed at Carnegie Mellon provides applications with an interface to their execution environment, which includes the network state. Remos is designed for the development of network-aware parallel and distributed applications on diverse network architectures; portability across network architectures and network installations were among the key design considerations. Efficiency is another important consideration, and the cost that an application pays in terms of runtime overhead is low and directly related to the depth and frequency of its requests for network information. The Remos system has its roots in two separate projects: an investigation of resource management in application-aware networks (Darwin) and an effort to support system-aware applications through libraries, frameworks, and tools (Remulac).

Remos allows network-aware applications 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 executing application. To avoid dependences on the idiosyncrasies of particular network architectures and communication systems, application development for networks requires a system-independent interface between applications and networks. Remos provides a uniform interface that allows the development of portable applications that can adapt on a range of network architectures. A system-independent interface is also crucial for allowing applications to adapt on heterogeneous networks, where subnets are realized with different network technologies.

Clusters of workstations are an increasingly popular platform for parallel computing but suffer from the problem that sharing of network resources can lead to poor and unpredictable performance. For many synchronous parallel applications, the communication performance is determined by the speed of the *bottleneck* link, and therefore, if even one communication link is busy, the application performance can degrade dramatically. Remos can help in solving this problem by reporting network conditions to applications, and thus allowing applications to adapt dynamically to network conditions. We present the framework for developing network-aware parallel applications with Remos and present preliminary results. In particular, we demonstrate that Remos driven node selection is important and effective for parallel computing on a shared high-speed network.

2 Usage models

Remos targets networked systems that consist 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 computing environment. We summarize some scenarios in parallel and distributed computing that can exploit network status information provided by Remos.

- *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 a variable 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 into the physical memory of all participating nodes. Many large-scale scientific simulations fit this pattern, for instance earthquake modelling (the Quake project [2] at Carnegie Mellon) and pollution modeling (Airshed [23]).

- *Application migration:* Changes in the environment may render the initial selection of nodes and connections unattractive. If the application is designed appropriately, the application mapping can be modified during execution to take the changes into account. The Fx programming environment at Carnegie Mellon [25] allows dynamic mapping of programs, and one goal of this research is to provide Fx with network information to drive mapping and migration.

- *Optimization of communication:* Closely related to the application mapping issues is the problem of exploiting low-level system information, such as network topology. As an example, if an application relies heavily on broadcasts, some subnets (with a specific network architecture) may be better platforms than others. This aspect is particularly important for heterogeneous networks.

- *Application quality metrics:* Some applications must 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.

- *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, e.g., when deciding whether to perform a simulation locally or on a remote server.

In the above usage examples, applications will in general have to be aware of all system resources, including both network and computation resources. The Remos interface focuses on providing applications with information about networking resources. System-aware applications typically require information about computation and memory resources available on the network nodes, and there exist a variety of approaches to obtain such information for different time windows (e.g., a real-time operating system may maintain information about immediate resource usage, or a database of historical load information can be used [5]). The focus of this paper is the use of network information, so we do not pursue the issue of dealing with other system resources further. However, Remos does include a simple interface to computation and memory resources.

The Remos interface does not include communication *operations*, only information about communication performance. Many communication frameworks, such as PVM [9] and MPI [7], provide a wealth of communication primitives, so there is no need to duplicate this aspect of parallel computing. Remos can be used in conjunction with such communication libraries, e.g., to optimize primitives in a communication library by customizing the implementation of group communication operations for a particular network.

3 Remos design challenges

In this section we briefly summarize the problems that a uniform, portable network interface like Remos must address:

- *Dynamic behavior*: Network conditions can change quickly, so the interface must characterize this dynamic aspect. Different applications may want this information in different formats, e.g., some applications are more interested in burst bandwidth while others primarily care about long term average bandwidth. Furthermore, applications are most interested in future network behavior not historical information, and hence it is important to provide future information, particularly when a reasonable basis for prediction (e.g., existing reservations) exist.

- *Sharing (internal and external)*: Connections (as seen by the application) share physical links with other connections. This dynamic sharing of resources with entities external to an application is the major reason for the variable network performance experienced by applications. Parallel and distributed computations often simultaneously transfer data across multiple point-to-point connections, so there is also internal sharing as these connections compete with each other for resources. An interface that provides performance data must therefore consider the network's sharing policy when estimating bandwidth availability.

- *Information diversity*: Applications may be interested in a wide variety of information, ranging from static network topology, dynamic bandwidth estimations on a variety of times scales, and latency information. This information may have to be retrieved through diverse mechanisms, ranging from querying information databases using standard protocols, to direct measurement for the network interface.

- *Network heterogeneity*: Network architectures differ significantly in their ability to collect information and make it available. The kind of information that is available from the network architecture impacts the design and overheads associated with a query interface.

- *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. The simplest solution seems to be to expose as much information as possible. However, such a solution conflicts with other goals, particularly the portability of the API across heterogeneous networks. It also raises both ease of use and scalability concerns (the Internet is too huge for exposing all information). An alternative is to provide the information at a much higher level, with focus on performance characteristics of interest to the application. A high level of abstraction can make the interface easier to use and avoid information overload, but it can also result in vagueness and inaccuracies.

- *Accuracy and fidelity:* The information provided by such an interface may not be completely accurate, e.g., the estimates of the future bandwidth may not take all traffic into account. Nor may the information be of high fidelity, e.g., if the average bandwidth over a large interval is computed based on a small number of measurements. The issue then is how much accuracy and fidelity should be demanded (from the network) or promised (to the application).

The challenge for the interface designer is to balance practicality and elegance. These challenges affect both the definition of the interface and its implementation(s). In the next section, we describe the design decisions. Our goal is to support high-level network-independent parallel programming, so we must ensure that the requirements of such programs are satisfied.

4 Remos API 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.

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.

A general flow query has the following form:

```
remos_flow_info(fixed_flows,
                variable_flows,
                independent_flow,
                timeframe)
```

Remos tries to satisfy the `fixed_flows`, then the `variable_flows` simultaneously, and finally the `independent_flow`. The corresponding data structures will be filled in to the extent that the flow requests can be satisfied. By combining flows in these three flow classes, 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 [14], which is the basis of ATM flow control for ABR traffic [16, 1], and is also used in other environments [12].

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.

The basic network topology query has the following form:

```
remos_get_graph(nodes,
                graph,
                timeframe)
```

The query fills a `graph` that is relevant in connecting the `nodes` set. The information annotated to the graph is relevant for the specified `timeframe`.

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

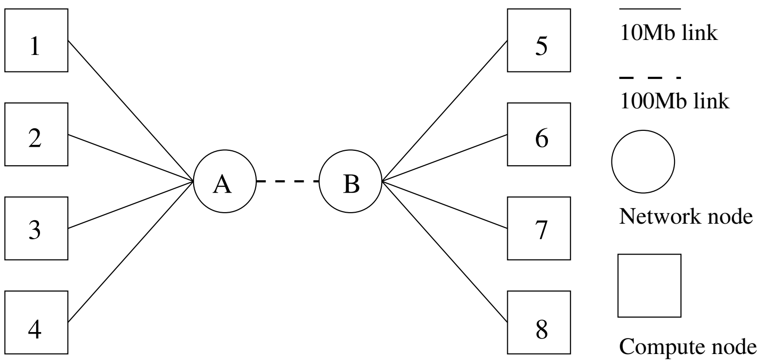


Figure 1: Remos graph representing the structure of a simple network. Nodes A and B are network nodes, and nodes 1–8 are compute nodes.

may not correspond to the physical structure of the Ethernet wiring, it accurately represents its performance.

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 application 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. However, in reality network information like bandwidth and latency is 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, we present the variability of network parameters using quartiles, which is considered the best choice for an unknown data distribution [15].

4.5 Limitations

Our design of Remos ignores a number of important network properties. It currently 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.

5 Implementation

The Remos implementation has two components, a Collector and Modeler; they are responsible for network-oriented and application-oriented functionality, respectively. This is illustrated in Figure 2. A Collector consists of a process that retrieves raw information about the network. The Collector that is used in experiments described later in the paper uses SNMP [4] to extract both static topology and dynamic bandwidth information from the routers. We also have a Collector that uses benchmarks to probe networks that do not respond to our SNMP queries (e.g. wide-area networks run by commercial ISPs). For latency, the Collector currently assumes a fixed per-hop delay. (A reasonable approximation as long as we use a LAN testbed.) The Collector is implemented in Java, since we envision that in the future it may be down-loaded across the network. A large environment may require multiple cooperating Collectors. The Modeler is a library that can be linked with applications. It satisfies application requests based on the information provided by the Collector. The primary tasks of the modeler are as follows: 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. The modeler exports Remos information through a Java and a C interface.

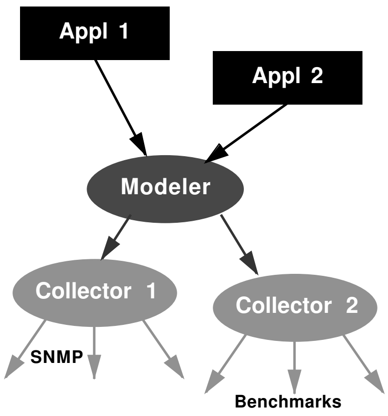
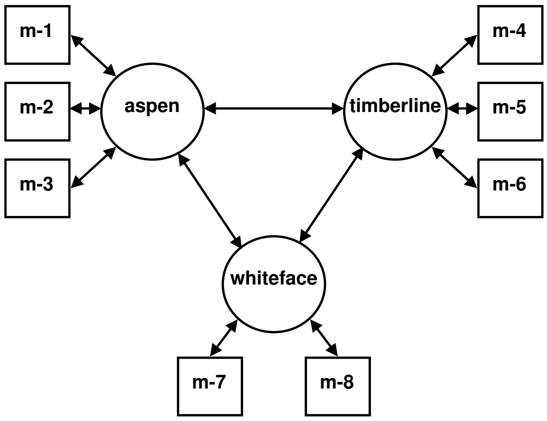


Figure 2: Remos implementation

This prototype Remos implementation has been tested on a dedicated IP-based testbed at Carnegie Mellon, a portion of which is illustrated in Figure 3. The testbed uses a variety of routers (PCs running NetBSD and FreeBSD and Cisco routers), 100Mbps and 10Mbps point-to-point Ethernet segments as links, and DEC Alpha systems, PCs running NT, PCs running FreeBSD as endpoints. An important advantage of using SNMP is that it is widely supported implying that the implementation is portable to many network environments. The portability is illustrated by the diversity of routers in our testbed.

We would like to point out that the Remos API is independent of its implementation, and suitable implementation techniques will be different for networks of different types and sizes, e.g. we use a Collector that use benchmarks to probe the network for environments where the use of SNMP is not possible or practical. We are also looking into the problem of dealing with very large networks, where multiple collectors will have to collaborate to collect the network information.



Links: 100Mbps point-to-point ethernet
Endpoints: DEC Alpha Systems (*manchester-** labeled *m-**)
Routers: Pentium Pro PCs running NetBSD *(aspen,timberline, whiteface)*

Figure 3: Testbed used for Remos experiments

6 Parallel application development with Remos

In this section we examine how Remos can help in meeting specific challenges in parallel computing in a networked environment. We first discuss the adaptation process and sketch some of the design choices for network-aware parallel programs. In the following sections, we discuss how a specific parallelizing compiler and runtime system (the Fx tool chain [11, 25]) were enhanced to support the development of Remos driven adaptive parallel programs and present experimental results demonstrating how programs adapt to changing network conditions.

A network-aware parallel application typically consists of a computation module and an adaptation module. The computation module implements the core functionality of the application, that is, its structure is similar to that of the complete application in a dedicated environment. The adaptation module collects information about the status of the network and decides how to best organize execution. In general, only the adaptation module interacts with tools like Remos that collect and deliver network information. Execution is controlled by setting specific parameters such as the number of nodes or the granularity of the computation between communication phases. An important goal of the adaptation module is to determine values for these "adaptation parameters" that optimize application performance (e.g. minimize execution time) given the current network conditions. Making this decision often requires some information from the computation module, e.g. input data sizes. The adaptation step can be very complex and application-specific, although support can often be provided through frameworks[3].

One of the main design decisions for network-aware applications is what parameters to adjust based on network conditions. Most previous work in this area has focused on video streaming applications. Video streaming has the property that the parameters to adjust to changes in the network are fairly obvious (typically the frame rate or frame size), can be adjusted easily, and appropriate values can be selected fairly easily. e.g. if the available bandwidth drops, the frame rate should be reduced. In contrast, parallel computing applications tend to be much more complex. They have a wide variety of parameters and the relationship between these parameters and application performance can be difficult to characterize. For example, the optimal number of nodes for a parallel computation often depends on network performance, but characterizing this relationship precisely is challenging [26]. Alternatively, the adaptation module may select which nodes should be used from a larger pool of available nodes, which is the problem we address in our experiments. Finally, the adaptation parameter may be internal to the application. For example, in [21] an adaption modules selects the optimal pipeline depth for a pipelined SOR application based on network and CPU performance.

Some existing programming tools are well suited to support network-aware adaptation [21]. Many tools such as parallelizing compilers and distributed object libraries already have information on the application properties which may be useful in selecting the parameters that are most sensitive to application performance, at what points during execution is adaptation legal and potentially profitable, or what network properties is the application most sensitive to (e.g., bandwidth or latency). Many tools in fact already optimize application performance based on system properties. parallelizing compilers often map the application considering system characteristics [24] and load balancing systems manage computations based on CPU loads on the different hosts in an environment. The contribution of this research is to allow consideration of communication load under dynamic network conditions, which is a fairly natural extension. Finally, one of the critical steps in using an interface like Remos is the specification of the communication pattern. Programming tools often have this information either because they generate the communication statements (e.g. parallelizing compilers) or because they use specific built-in traffic patterns (e.g. object libraries).

Adaptation to network conditions using Remos is a three step process. First, the parallel application has to formulate its computation and communication requirements and specify the network parameters and nodes that are of interest. Second, Remos is invoked to collect the network information. Finally, the application adapts by selecting appropriate runtime parameters. The easiest time to take network conditions into account is when the application starts up, although applications can adapt periodically at runtime in response to changes in the environment. Adaptation at runtime can be more complex since it often requires the use of more complex data structures that can be dynamically redistributed [20]. Moreover, runtime adaptation with distributed data structures is possible only if the state of the computation can be redistributed across the nodes. This can be expensive, and this overhead has to be considered when evaluating adaptation options, thereby complicating the adaptation process.

7 Remos usage framework

In this section we describe a framework in use at Carnegie Mellon for developing adaptive parallel programs, along with a set of applications developed in this framework. The tool-chain consists of Remos, Fx compiler system [11, 25], and a clustering module. The goal is that the applications should have the ability to migrate to different parts of the network in response to changes in the network traffic. In the next section we discuss the results and experience with developing applications in this framework.

7.1 Fx compiler

The Fx compiler system developed at Carnegie Mellon supports integrated task and data parallel programming. The user writes a sequential program with annotations to assist parallelization, and the compiler generates the programs for the nodes of the parallel system. Fx is based on a variant of the HPF language [17].

The Fx runtime system was enhanced so that the assignment of nodes to tasks in a program could be modified during execution. In this model, a program is invoked on all nodes that may be potentially used for execution. The *task parallelism* support in Fx is used to map the core computation onto an *active task*. The set of processors assigned to the active task can be changed at runtime, thereby effectively migrating the program. The nodes can be reassigned without changing the semantics of the program at any program point where there is no live distributed data. Note that live distributed data can be copied as replicated data to create legal migration points. This step can be expensive in terms of memory usage and copying time, but it makes it possible to create legal migration points in programs with no migration points or, insufficient frequency of migration points to support effective network-aware execution.

7.2 Clustering

An important problem in developing a network aware application is selecting a set of nodes that are well connected in terms of available communication resources. The goal is to select nodes with low latency and high available bandwidth between them. The bandwidth and latency measurements can be obtained from Remos, but the problem of determining the optimal set of nodes is computationally hard [1], which is especially a cause for concern for runtime migration.

For the purpose of our experiments, we have chosen a simple greedy heuristic for clustering. The application provides an initial *start node*, which is the first node that is added to the selected *cluster* of nodes. Next , the node with the *shortest distance* to the existing nodes in the cluster is determined and added to the cluster. The basis for *distance* between a pair of nodes is the current latency and bandwidth between the nodes, which is obtained from Remos. The above step is repeated until the cluster contains the number of nodes needed for execution.

We will show in the next section that this clustering procedure leads to good results even though it is based on a simple heuristic. However, it is clear that improved procedures will be needed to achieve good and fast clustering in more complex situations. In particular, we have focused on communication resources, but in general, tradeoffs between computation and communication resources would have to be considered for clustering.

7.3 Application structure

We describe the procedure for developing a network-aware parallel program that can migrate in response to changing network conditions. We restrict ourselves to iterative applications that adapt (if necessary) at the beginning of every iteration of an outer loop. At these migration points, the application calls an adaptation module with parameters for migration, and this adaptation module determines the potential profitability of migration and performs migration if necessary.

When the adaptation module is invoked, it checks if Remos is active on the set of nodes the application is interested in, and starts it up if needed. Next, it calls a Remos routine to obtain the logical topology of the relevant graph. For our experiments, this routine `remos_get_graph` is called for the set of nodes in the testbed shown in Figure 4, and returns the logical network topology connecting these nodes, along with available network bandwidth on the links based on current traffic conditions. We state this call to `remos_get_graph`, along with a summary of the relevant parameters.

```
remos_get_graph(nodes, graph, timeframe)
```

`nodes` = **m1,m2,m3,m4,m5,m6,m7,m8**
`graph` = *logical topology of the testbed shown in Figure 4 with current traffic*

[1]It is equivalent to a k-clique problem which is known to be NP-hard.

`timeframe` = **current:** *most recent measurements are used*

The logical topology graph is used to compute a matrix representing *distance* between all pairs of nodes. For our testbed, the distance is based only on bandwidth since latency between any pair of nodes is virtually the same. Note that the information to compute available bandwidth between pairs of nodes could have been obtained with flow queries also, but $O(nodes^2)$ queries would have been needed, implying a much higher overhead which deteriorates rapidly for larger networks.

The communication distance matrix, the number of nodes required for the application, and an application specified initial node are the inputs to the clustering routine discussed earlier. The clustering routine returns the estimated optimal set of nodes for execution along with a measure of the expected communication performance. This communication performance is compared to the estimated performance on the current set of executing nodes, and if the potential improvement is above a specified threshold, the application is migrated to the new set of nodes by providing a new list of nodes to the Fx runtime system.

8 Usage examples and experimental results

We present a preliminary report on the usage of the Remos framework in the context of parallel programs generated by the Fx compiler [11, 25]. We present results from network-aware versions of the following two programs: *fast Fourier transforms (FFT)* and *Airshed pollution modelling*. The FFT program performs a two dimensional FFT, which is parallelized such that it consists of a set of independent 1 dimensional row FFTs, followed by a transpose, and a set of independent 1 dimensional column FFTs. Airshed contains a rich set of computation and communication operations, as it simulates diverse chemical and physical phenomena [23]. Both are data parallel programs developed using the Fx compiler system. All experiments were performed on the IP-based testbed illustrated in Figure 4.

The remainder of this section is organized as follows. We first present results that highlight the importance of node selection in an unloaded network, followed by similar results for networks with communication loads. Finally we discuss the dynamically migrating implementations of these applications.

8.1 Node selection in a static environment

We first examine the value of node selection when there are no competing applications on the testbed. We should point out that our testbed presents a difficult test for the clustering routine using

Remos. Since the testbed is connected with high performance links and switches throughout, and any node can be reached from any other node with at most 3 hops, it is difficult to judge what sets of nodes are better connected than others.

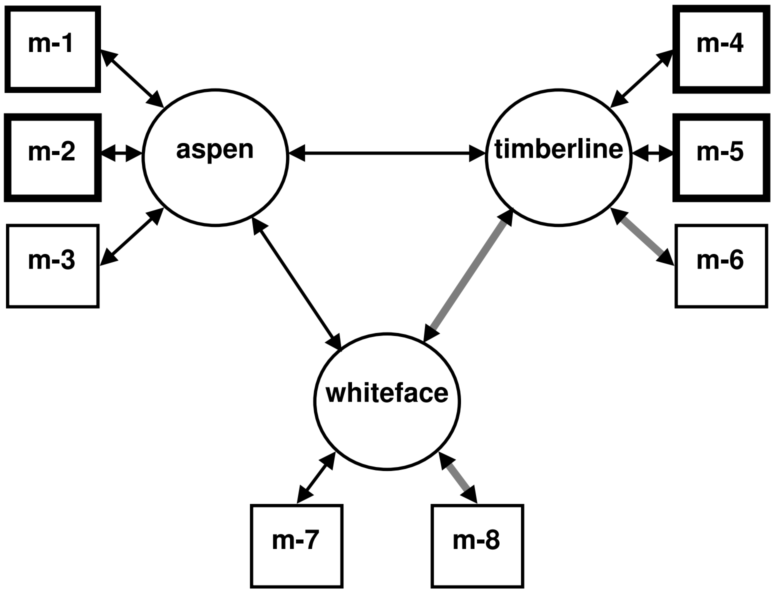
The FFT program and the Airshed program were developed in the framework described in the previous section and executed on sets of nodes selected by Remos based clustering at the time of invocation. For comparison, they were also executed on other representative sets of nodes. In all examples, the programs gave node *m-4* as a start node and used clustering to select remaining nodes. The results are presented in Table 1. We observe that the execution time on the Remos selected nodes was generally (but not always) lower than for other node sets, but only by relatively small amounts. As noted earlier, this is a hard case for Remos based clustering, but our toolchain is doing a good job of node selection. However, we note that node selection has limited importance on a small network with fast routers and links and no external traffic.

Problem Description		Remos Selected Nodes		Other Representative Node Sets					
Name of Program	No. of Nodes	Node set	Exec. Time (secs)	Node set	Exec. Time (secs)	Percent increase	Node set	Exec. Time (secs)	Percent increase
FFT (512)	2	m-4,5	.462	m-1,4	.468	1.3	m-4,8	.481	4.1
FFT (512)	4	m-4,5,6,7	.266	m-1,2,4,5	.287	3.7	m-1,4,6,7	.268	.03
FFT (1K)	2	m-4,5	2.63	m-1,4	2.66	1.1	m-4,8	2.68	1.9
FFT (1K)	4	m-4,5,6,7	1.51	m-1,2,4,5	1.62	7.3	m-1,4,6,7	1.61	6.6
Airshed	3	m-4,5,6	908	m-4,6,8	907	-.1	m-1,4,7	917	1.1
Airshed	5	m-4,5,6,7,8	650	m-1,2,3,4,5	647	-.4	m-1,2,4,5,7	657	1.1

Table 1: Performance of programs on nodes selected using Remos on our IP based testbed

8.2 Node selection in a dynamic environment

To study the importance of node selection in the presence of competing applications, we performed another set of experiments. We added a synthetic program that generates communication traffic between nodes *m-6* and *m-8* on our testbed, and performed node selection in the presence of this traffic. The process is illustrated in Figure 4. The figure shows our testbed with communication traffic between nodes *m-6* and *m-8*. The links that carry this traffic are highlighted. The node selection procedure is given *m-4* as the start node, and it selects nodes *m-1,m-2,m-4,m-5* as the cluster most appropriate for execution. Note that this is one of the sets for which the application traffic does not interfere with the external traffic on the network. Selection of either of nodes *m-6,* *m-7* or *m-8* would imply that at least one link is shared between the application traffic and the external traffic.



Traffic Route: m-6 -> timberline -> whiteface -> m-8

Start Node: m-4

Selected Nodes: m-1, m-2, m-4, m-5

Figure 4: Selection of nodes on the testbed with busy communication links

For our measurements, we first executed the program on Remos selected nodes as discussed above. For comparison, the programs were re-executed on a set of nodes that the clustering routine could have potentially selected if it only used the static physical capabilities of the testbed communication nodes and links and ignored the traffic The results are presented in Table 2.

We observe that the execution times are larger by 80-200 percent when dynamic traffic information is not used for node selection. The reason is that the selection of nodes using Remos measured dynamic traffic makes it possible to avoid links with heavy traffic. The table also shows the performance of the programs in the absence of the external traffic (last column). We observe that with our dynamic node selection, the performance degrades only marginally in the presence of traffic, but can degrade dramatically for a naively selected set of nodes. The conclusion is that node selection is very important for good performance of data parallel programs in dynamic environments with competing traffic, and that our clustering routine using Remos is effective in selecting good sets of nodes in such realistic situations.

Problem Description		Execution Time with External Traffic					Exec. time without External Traffic (secs)
		Remos selected Nodes (Dynamic Measurements)		Nodes selected with only Static Measurements			
Name of Program	No of Nodes	Node Set	Time (secs)	Node Set	Time (secs)	Percent increase	
FFT (512)	2	m-4,5	.475	m-4,6	1.40	194	.462
FFT (512)	4	m-1,2,4,5	.322	m-4,5,6,7	.893	177	.266
FFT (1K)	2	m-4,5	2.68	m-4,6	7.38	175	2.63
FFT (1K)	4	m-1,2,4,5	2.07	m-4,5,6,7	3.71	79	1.51
Airshed	3	m-1,4,5	905	m-4,5,6	2113	133	908
Airshed	5	m-1,2,3,4,5	674	m-4,5,6,7,8	1726	156	650

Table 2: Performance implications of node selection using Remos in the presence of external traffic. Measurements use a synthetic program that generates significant traffic between nodes m-6 and m-8 on our IP-based testbed

8.3 Runtime adaptation

We have also developed a set of applications that periodically select a new set of nodes for execution and migrate to those nodes dynamically. This process is significantly more complex than selection of nodes at the start of execution, as the program state has to be migrated. For easing the process of adaptation, we assume that the active data set at migration points is replicated and all previous communication has completed. Hence, no data copying or explicit synchronization is necessary for migration.

Our observations supported the intuitive expectation that executing applications would automatically migrate to the least loaded part of the network. A general validation of migration is complex and beyond the scope of this paper. The main reason is that the results strongly depend on the traffic patterns during execution. Here we present results from a simple experiment with an adaptive version of the Airshed simulation in different static network traffic scenarios. The program was compiled for 8 nodes but only 5 nodes effectively participated in the computation. The mapping of these 5 *active* nodes was automatically changed such that the program would always execute on the part of the network with the least amount of traffic. Timing measurements were made on the testbed without traffic, with traffic that did not interefere with the starting nodes for the computation, and with traffic that did interfere with the computation. The results are listed in Table 3.

In the absence of network traffic, we observe that the adaptive version of Airshed programmed

Executing Node Set	Execution time with traffic patterns (seconds)			
	No Traffic	Non-interfering Traffic	Interfering Traffic-1	Interfering Traffic-2
Fixed	862	866	1680	1826
Adaptive	941	974	1045	955

Table 3: Execution times of adaptive version of Airshed executing on a fixed set of nodes and on dynamically selected nodes. The non-adaptive Airshed executes in about 650 seconds

to run on a fixed set of nodes took around 862 seconds, which is significantly slower than the non-adaptive version (that took around 650 seconds as discussed in earlier sections). This represents the overhead of compiling for 8 nodes and running on 5 nodes, which is an artifact of the model of adaptive programming that was used. When active adaptation was in force in the absence of traffic, the average execution time was 941 seconds, which is moderately higher than the case when no adaptation was done (862 seconds). This difference reflects two types of overheads of adaptation. First, there is a cost involved in analyzing and deciding the best nodes for execution. Second is the cost of unnecessary program migration which can happen because of the imprecise nature of the decision making mechanism for adaptation. This aspect is dicussed later in this section. Finally, in the presence of interfering network traffic, the adaptive version of Airshed performs significantly better than the version that runs on a fixed set of nodes. The general conclusion is that support for adaptation can entail moderate overheads, but it can dramatically minimize the impact of external traffic on execution time in many scenarios.

There are a number of issues that are important for runtime adaptation. The application framework has to determine the minimum potential improvement in performance before deciding to migrate. In our experiments, migration was done whenever the potential improvement was positive. One undesirable consequence that was observed was that the program would often migrate between sets of nodes that were equally desirable for execution. A more insidious problem is that the traffic generated by the application would make some links busy, and since Remos does not distinguish between different types or sources of traffic, the result would be that the application would migrate to avoid its own traffic, which is clearly a decision based on an inherent fallacy. This can be avoided if the application knows how much communication traffic it generates and factors that into making migration decisions.

9 Related Work

A number of resource management systems allow applications to make queries about the availability of computation resources, some examples being Condor [18] and LSF (Load Sharing Facility). In contrast, Remos focuses on network resources. More recently, resource management systems have been designed for large scale internet-wide computing, e.g., Globus [8] and Legion [10].These systems provide support for a wide range of functions such as resource location and reservation, authentication, and remote process creation mechanisms. The Remos interface focuses on providing good abstractions and support for giving applications access to network status information, and allow for a much closer coupling of applications and networks. The Remos interface could be used to extend the functionality of the above systems.

A number of groups have looked at the benefits of explicit feedback to simplify and speed up adaptation [13, 6]. However, the interfaces developed by these efforts have been designed specifically for the scenarios being studied.

A number of sites are collecting Internet traffic statistics. This information is not a in form that is usable for applications, and it is typically also at a coarser grain than what applications are interested in using. Another class of related research 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 [22].

Parallelizing compilers and runtime systems attempt to balance load across the processors using a variety of adaptive techniques for partitioning computation and communication, an example being [19]. However, such techniques are generally limited to the nodes pre-selected for the application and do not involve other nodes in the system.

10 Concluding remarks

Remos is an example of a performance monitoring interface that allows parallel applications to inquire about the capabilities of their host network. Remos provides a uniform interface between networks and network-aware applications that covers a range of network architectures. It allows applications to discover, at runtime, the properties of their execution environments. Such interfaces are of great value to compilers and tools that manage placement of computations onto network nodes, or that attempt to maximize performance. The topology and behavior of networks will change from application invocation to invocation and may even change during execution. Hence, it is not practical for the compiler to model the network, so any attempt to optimize performance based on network conditions must be based, at least in part, on runtime information. The Remos interface allows tools and applications to obtain the information they need to make performance optimizations.

An initial implementation of the Remos API demonstrates that the interface can be realized and our initial evaluation using a small set of applications suggests that applications can easily use the interface and benefit from the the information it provides. In this paper we focused on parallel applications that were developed using parallel programming tools, specifically the Fx compiler and runtime system. In such an environment, the runtime library can use Remos to collect information about network conditions, and simple network aware adaptation decisions can be performed by the tools in a transparent fashion. However, more research is needed to establish the role of such adaptation support tools for different types of application and network behavior.

A uniform interface for network-aware applications significantly reduces the difficulty of coupling an application to a network. As network architectures continue to evolve, we anticipate that interfaces like Remos will play an increasingly important role in raising the acceptability and practicality of network-aware applications.

References

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

[2] 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).

[3] BOLLIGER, J., AND GROSS, T. A framework-based approach to the development of network-aware applications. *IEEE Trans. Softw. Eng. 24*, 5 (May 1998), 376 – 390.

[4] 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.

[5] DINDA, P. Statistical properties of host load in a distributed environment. In *Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers* (Pittsburgh, PA, May 1998).

[6] ECKHARDT, D., AND STEENKISTE, P. A Wireless MAC with Service Guarantees. In preparation, 1998.

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

[8] FOSTER, I., AND KESSELMAN, K. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications 11*, 2 (1997), 115–128.

[9] 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.

[10] GRIMSHAW, A., WULF, W., AND LEGION TEAM. The Legion vision of a worldwide virtual computer. *Communications of the ACM 40*, 1 (January 1997).

[11] GROSS, T., O'HALLARON, D., AND SUBHLOK, J. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology 2*, 3 (Fall 1994), 16–26.

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

[13] 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.

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

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

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

[17] KOELBEL, C., LOVEMAN, D., STEELE, G., AND ZOSEL, M. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[18] 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).

[19] SHARMA, S., PONNUSAMY, R., MOON, B., HWANG, Y., DAS, R., AND SALTZ, J. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing '94* (Washington, DC, Nov 1994), pp. 97–106.

[20] SIEGELL, B. *Automatic Generation of Parallel Programs with Dynamic Load Balancing for a Network of Workstations*. PhD thesis, Department of Computer and Electrical Engineering, Carnegie Mellon University, 1995. Also appeared as technical report CMU-CS-95-168.

[21] 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.

[22] 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).

[23] 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).

[24] 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.

[25] 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.

[26] 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.