Invited paper at the 8th Heterogeneous Computing Workshop (HCW '99)

Adaptive Distributed Applications on Heterogeneous Networks

Thomas Gross[1,2], Peter Steenkiste[1] and Jaspal Subhlok[3]

[1]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[2]Departement Informatik
ETH Zürich
CH 8092 Zürich

[3] Department of Computer Science
University of Houston
Houston, TX 77204

Abstract

Distributed applications execute in environments that can include different network architectures as well as a range of compute platforms. Furthermore, these resources are shared by many users. Therefore these applications receive varying levels of service from the environment. Since the availability of resources in a networked environment often determines overall application performance, adaptivity is necessary for efficient execution and predictable response time. However, heterogeneous systems pose many challenges for adaptive applications. We discuss the range of situations that can benefit from adaptivity in the context of a set of system and environment parameters. Adaptive applications require information about the status of the execution environment and heterogeneous environments call for a portable system to provide such information. We discuss Remos (Resource Monitoring System), a system that allows applications to collect information about network and host conditions across different network architectures. Finally, we report our experience and performance results from a set of adaptive versions of Airshed *pollution modeling application executing on a networking testbed.*

1 Introduction

Many distributed applications have critical response time requirements. The timeliness of a response however depends on the availability of resources: network bandwidth to transfer information and processor cycles to perform computations. In heterogeneous environments, applications seldom have exclusive access to resources. Instead, network links and processors are shared by many applications and users.

The performance of a fast processor or network link can deteriorate to that of a slow one with additional computation load, but if the application can move to another system, then the user may not experience a slowdown. When running a distributed simulation, the impact of link congestion can be avoided by migrating to a different part of the network. A data warehouse may appear to stop operating when additional users start expensive queries, but if the data is replicated on another server, the application may switch to this server and thereby preserve the perception of a timely response. The transfer of a movie is subject to many dropped frames if there is network congestion. However, a smart filter may be able to remove non-essential frames from the movie and maintain audio and video synchronization by reducing bandwidth requirement.

All of these examples of adaptivity have been explored in various systems. In this paper we attempt to present a structure to these approaches that allows us to unify the development of interfaces between applications and environments. Since heterogeneous environments provide many challenges to application developers, it is important that the interface that provides network measurements is simple and portable. We believe that a uniform framework for developing adaptive applications and resource monitoring systems that work across different network architectures are the essential ingredients for speeding up the development of adaptive applications.

The remainder of this paper is organized as follows. We first describe the "space" of adaptation options that are available, using an example scientific simulation to illustrate the choices. We then give an overview of Remos system for collecting and reporting network status, and present performance results for an adaptive environmental modeling application. We conclude with a discussion of related work.

2 Adaptivity of applications

Adaptivity allows applications to run efficiently and predictably under a broader range of conditions. Support for adaptation may also allow applications to use less expensive service classes, e.g. best effort instead of guaranteed service. Some of the functionality (and complexity) associated with adaptation can be embedded in middleware, but we first have to understand the dimensions of adaptation before we can develop general purpose libraries or middleware layers to support adaptivity.

Applications can adapt along a number of "dimensions". In this paper we focus on the choice of resources (space dimension), the time of adaptation, and the interface between the application and the runtime system (or operating system, i.e., the system that is responsible for management of resources). In each case we first sketch the full spectrum of options available to applications in general, and we then focus on the options that are of most interest to distributed scientific simulations. We use Airshed environment modeling application described in Section 4 to illustrate the performance tradeoffs associated with adaptation.

2.1 Resource classes

Applications have access to a wide range of resources, and they often have a choice about how many and which resources they can use. An application can be adaptive with regard to the *number of processors* or nodes that it can use or its adaptivity may be restricted to the *space* of the network environment, i.e., the number of nodes is fixed but the identity of the nodes is determined dynamically.

Network resources are another candidate for adaptivity. Network bandwidth can sometimes be traded off with other parameters such as the fidelity of the data or the quality of the objects that are transferred. For example, by changing the size or the frame rate of a movie, an application can increase or decrease its bandwidth demands. Alternately, applications can make tradeoffs between different types of resources, e.g., compression can be used to reduce the bandwidth requirements, but then CPU cycles are required to compress and decompress the data.

Network resources are often not directly accessible to an application but their use is determined by the kind of service that the application requests. Recently, the networking community has been working on developing integrated services networks that can offer a range of services [4]. The *service class* dimension reflects the fact that an application can pick a service class that best matches its needs. This decision may (should) be based on dynamic conditions. E.g., when setting up a video conference over a network that supports differentiated service, the user or the application would like to pick the lowest service class (best effort service) that can provide sufficient bandwidth. A higher service class (e.g., expedited service) will be selected only if it can deliver the bandwidth that a lower service class is unable to do.

Scientific simulations can potentially use any of the above methods. The most common form of adaptation along the *space* dimension is likely to be the addition or deletion of execution nodes, as well as migration to a different subnet for execution. Rebalancing the load on different nodes and links can be used as a mechanism to adjust to the changing network status. Another option for adapting is modifying the mapping style of the computation onto the nodes, e.g., replication of data and computation to eliminate communication. In some cases application components can choose between multiple algorithms with different computation and communication requirements, and they can switch from one to the other when network conditions change. Finally, scientific simulations can also adapt in the *service class* dimension in a variety of ways, although relatively few networks today offer more than one service class.

2.2 Time of adaptation

Along the *time* dimension, at one extreme, applications adapt only at compile time. E.g., the user may hardwire the number of processors (nodes) into an application by specifying this number at the time the application is compiled. However, this scenario hardly qualifies as "adaptivity", so we will not discuss it further.

A more flexible option is that the program be compiled for a variable number of nodes and the actual number of nodes for execution is determined at the time the application is executed. Adaptation is in general based on the assumption that recent past conditions are a good predictor of near-term future conditions, an assumption that often holds. Dynamic adaptivity provides the most flexibility but also poses the biggest challenges to the application designer. The designer has two options with different benefit/complexity tradeoffs. One option is to limit adaptation to *load* or *start-up* time. This option is the easiest one since the applications has not set up any state yet. It has the obvious drawback that if conditions change during execution, the application will be unable to adapt to those changes. An example is an application that has a choice about what nodes, and thus what part of the environment, to use. A Web browser may be able to choose from several replicated servers or a proxy cache. An alternate model is to allow the application to adapt not only at startup but also at runtime. Such behavior is more complex to implement since it means that the application must be able to reconfigure itself. This capability requires changes in the application state and compute environment and therefore typically does not exist in today's applications; it must be added to make the application adaptive.

Runtime adaptation along the time dimension is addressed by protocols such as TCP and has also received the most attention from researchers studying network-aware applications. For applications that adapt dynamically, we can distinguish between applications that adapt periodically (e.g., a system that rebalances the loaded every k units of time) and systems that include *demand-* or *opportunity-driven* adaptivity. A system may adapt whenever some performance parameter drops below a threshold or may opportunistically attempt to utilize extra resources as they become available.

Distributed simulations can benefit from adaptation both at startup and at runtime. At startup, they typically have to decide on the number of nodes to use [25] and on the setting of some control parameters, e.g., pipeline depth [19]. At runtime, they can periodically re-evaluate their options, and adaptation may take the form of migration of the executing program to a different part of the network or rebalancing or remapping of the computation on the executing nodes. This runtime adaptation adds considerable complexity to the program development and adaptation process, but is essential to get good performance for long running applications in dynamically changing conditions.

2.3 Information about the environment

To adapt, applications need information on the status of the environment. Traditionally, for network resources this task has been performed by communication protocols, such as TCP, so it is worthwhile to look at how these protocols collect information about network conditions. Protocols are often classified as using *implicit* or *explicit* feedback from the network. In protocols based on implicit feedback, the receiver monitors the incoming data stream and uses this stream to derive information about network conditions. TCP is a good example: dropped packets are viewed as a sign of congestion, and the sender responds by reducing its rate. In contrast, with explicit feedback, some entity inside the network provides explicit information about network conditions to senders. A good example is the ATM ABR traffic class: senders receive periodic information about network congestion conditions (e.g., congestion bit in rate management cells) or even the specific maximum traffic rate they are allowed to use (e.g., EPRCA).

Implicit feedback has the advantage that it does not require support from the network, so this approach to provide feedback is always feasible[1]. Implicit feedback also has some disadvantages: (i) it only allows incremental adaptation (i.e., when two hosts communicate, implicit information provides updates on how the bandwidth between these hosts evolves), and (ii) it is sometimes difficult to interpret the "information". (E.g., packet loss is an indication of congestion, but it is not always clear how the application should respond: pause for the duration of a round-trip time, reduce the congestion window, retransmit packets, etc.) Explicit feedback is in general easier to use, but it requires network support. Protocols today primarily rely on implicit feedback, and the same is true for most current network-aware applications. The reason is simple: implicit feedback does not require networking support, which does not exist.

Explicit information can be provided to the application in two ways. First, the network can provide feedback continuously. This approach is, e.g., employed for ABR traffic: a rate management cell is exchanged with the network for every 32 data cells. Continuous feedback is most often based on network properties that subsequently must be interpreted in the application space; for this reason this kind of coupling is also called *indirect*. An alternative is that applications receive a notification when specific events happen, e.g, an application receives an asynchronous notification when the network bandwidth drops below a certain threshold, or when the connection switches from one type of network to another [20]. Such notifications can be in the form of callbacks, or by invoking a specific event handler. With this style of interaction, the relationship between the network event (e.g., drop in bandwidth) and the actions (by the application or protocol software) is clearly established (e.g., when registering the handler). Therefore we call this style *event-driven* or direct coupling.

Scientific simulations can obtain network status information *externally* by using a tool to measure the activity on the network or *internally* by measuring the progress of work on different nodes and different parts of the network.

Simulations often use load balancing to improve the performance by giving less work to the nodes that are running slower than others. Load balancing can be implemented fairly well by internal measurements as the rate at which the work is progressing is a good indicator of the availability of resources. The general adaptation model, in which the application monitors its own performance and adapts when it observes a degradation (e.g. data loss), is widely applicable. It is possible to provide support for this form of adaptation through the use of frameworks [2] or other adaptation models [20].

However, other forms and dimensions of adaptation cannot be satisfied without external measurements. Selection of nodes at the the start of execution must be made with external measurements, as only those data points may be available at the time of invocation. Dynamic migration to a new set of nodes, as well as addition of nodes during execution, also requires external information, as internal information is limited to the current executing nodes.

Finally, it is useful to distinguish between the interface used by the application and the functionality supported by the network, since it is possible for a library or middleware layer to translate one interface into another. E.g., a library could translate continuous network feedback into event-based application feedback. A more interesting approach includes activities by the middleware: middleware could use a set of benchmarks to collect information on the conditions in a network (that does not provide explicit feedback) and present this information to the application in an explicit form. In this paper, we focus on the application level interface, and we only touch briefly on the lower level interface when we discuss implementation options.

[1] Implicit feedback typically makes some assumptions about the network, e.g., it considers packet loss to be a sign of congestion.

2.4 Network-application interactions

To be able to adapt along all three dimensions, applications will need information on network conditions, which span a matching space with the same dimensions. How applications collect network information determines how easily applications can explore this space.

Implicit information is based on experience, which severely restricts what part of the information space is explored: the application only learns about the part of the space it currently operates in. This means that it can collect information only on the network conditions along the paths it is currently using and on the service class it is in. Implicit feedback provides information along the time dimension, but only while the application is actively using the network. At startup or after the application has been idle for a while, no useful information is available.

We argue that given the limits on what information can be collected using implicit feedback, mechanisms must be provided so that applications can get explicit information on network conditions, e.g., by querying a standard interface. Such an interface should allow applications to collect information on network conditions in the entire network space (space, time and service class dimensions), allowing applications to make adaptation decisions in space, across service classes, and at startup.

One can argue that the restrictions on the type of information that can be collected using implicit feedback is not fundamental. Applications can use probing to explore the entire information space, e.g., they can periodically try all service classes and they can measure the network performance between every pair of usable hosts. While this approach may be appropriate in some cases, it is in general undesirable. First, developing effective network probing routines is difficult; it is not something that application developers should be required to do. Second, probing can be expensive, both in terms of elapsed time for the application and consumed network resources. Furthermore, excessive probing may disturb the measurements taken by this and other applications. In fact, large scale probing by applications would negate many of the advantages of implicit feedback. If probing is needed, it should be performed by a middleware layer. Then the probing code can be developed as part of the network architecture, and the collected information can be shared by many applications.

Note that we are proposing explicit feedback as a complementary mechanism to implicit feedback, and not as a replacement. Implicit feedback has clear advantages when used appropriately. Implicit feedback will remain useful as an inexpensive way of getting continuous feedback, once a particular operating point along the space and service dimensions has been selected. Implicit feedback is also likely to give more accurate and timely information (in a narrower part of the operating space) than explicit feedback.

3 A base system: Remos

The Remos API provides a query-based interface that allows clients to obtain "best-effort" information [14] on network conditions. The applications specifies the kind of information it needs, and Remos supplies the best available information. To limit the scope of the query, the application must select network parameters and parts of a larger network that are of interest. In this section we briefly describe the main Remos features. A more detailed description can be found elsewhere [14].

3.1 Level of abstraction

To accommodate the diverse application needs, the Remos API provides two levels of abstraction: high level flow-based queries and lower level topology-based queries.

Remos supports flow-based queries. A *flow* is an application-level connection between a pair of computation nodes. Queries about bandwidth and latency of sets of flows form the core of the Remos interface. Using flows instead of physical links provides a high level of abstraction that makes the interface portable and independent of system details. Flow-based queries place the burden of translating network-specific information into application-oriented information on the implementor of the API. However, flows are an intuitive abstraction for application developers, and they allow the development of adaptive network applications that are independent of the heterogeneity inherent in a network computing environment.

Remos also supports queries about the network *topology*. 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. E.g., finding the pair of nodes with the highest bandwidth connectivity is expensive using only flow-based queries. The topology information provided by Remos consists of a graph with compute nodes, network nodes, and links, each annotated with their physical characteristics, such as latency and available bandwidth. Topology queries return a *logical* interconnection topology. This means that the graph represents the network behavior as seen by the application, and does not necessarily reflect the physical topology. Using a logical topology gives Remos the option of hiding network features that do not affect the application. E.g., subnets can be replaced by (logical) links if their internal structure does not affect applications. Topology information is in general harder to use than flow-based information, since the complexity of translating network-level data into application-level information is mostly left to the user.

3.2 Dynamic resource sharing

Since networks are a shared resource, it is important to account for the manner in which resources are shared by multiple flows. Since multi-party applications use multiple flows, it is not only necessary to account for sharing across applications, but also across flows belonging to the same application. To be able to consider the effects of "internal" sharing, Remos supports multi-flow queries in which the application lists all its flows simultaneously. Applications can generate flows with very diverse sharing characteristics, ranging from constrained low-bandwidth audio to bursty high-bandwidth data flows. Remos collapses this broad spectrum into three types of flows. *Fixed flows* have a specific bandwidth requirement. *Variable flows* have related requirements and demand the maximum available bandwidth that can be provided to all such flows in a given ratio. (E.g., all flows in a typical all-to-all communication operation have the same requirements.) Finally, *independent flows* simply want maximum available bandwidth. These flow types also reflect priorities when sufficient resources are not available to satisfy all the flows. *Fixed flows* are considered first, followed by *variable flows*, then *independent flows*.

Determining how the throughput of a flow is affected by other messages in transit is very complicated and network specific. Remos approximates this complex behavior by assuming that, all else being equal, the bottleneck link bandwidth is shared equally by all flows (that are not bottlenecked elsewhere). If other 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 [11]. Applications that use topology-based queries are themselves responsible for taking the effects of both internal and external sharing into account.

3.3 Accuracy

Applications ideally want information about the level of service they can expect to receive in the future, but most users today must use past performance as a predictor of the future. 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 that allows the user to request data collected and averaged for a specific time window.

Network information such as available bandwidth changes continuously due to sharing and as a result, characterizing these metrics by a single number can be misleading. E.g., knowing that the bandwidth availability has been very stable represents a different scenario from it being an average of rapidly changing instantaneous bandwidths. To address these aspects, the Remos interface adds statistical variability and estimation accuracy parameters to all dynamic quantitative information. Since the actual distributions for the measured quantities are generally not known, we present the variability of network parameters using quartiles [12].

3.4 Implementation

An initial version of Remos API has been implemented. It has two components, a *collector* and *modeler*, that are responsible for network-oriented and application-oriented functionality, respectively. The collector is responsible for collecting low-level network information. Such data can be collected in many ways, e.g., one can periodically run benchmarks that probe the network for available bandwidth or rely on information gathered by applications [21]. Our current implementation uses a third method: the collector explicitly queries routers using SNMP [3] for both topology and dynamic bandwidth information. The use of SNMP to obtain information about the state of a network is a standard way of monitoring networks, and it should allow us to collect detailed information in a relatively non-intrusive way on a broad set of networks. The modeler is a library that is linked with the application; it translates the information provided by the collector into a logical topology graph or per-flow data in response to application requests. The modeler-collector architecture is in part motivated by the need to support scalability and network heterogeneity. In large networked environments, multiple collectors may have to be deployed, and each collector can collect information in a way that is most appropriate for the network it is responsible for. Work is in progress on implementing collectors that use sources of network information other than SNMP, e.g., by active measurements.

For the results presented in this paper we used the Remos interface on a dedicated IP-based testbed at Carnegie Mellon University that is illustrated in Figure 2.

4 Case study in adaptive execution: Airshed pollution modeling

We have developed a suite of tools to develop adaptive distributed programs driven by Remos and have gained experience with programs ranging from small kernels like fast Fourier transforms to complete applications like Airshed pollution modeling and magnetic resonance imaging[5]. In this paper, we focus exclusively on Airshed pollution modeling application and present results comparing the perfor-

	Explicit	Implicit
Indirect (continuous)	Queries to network management database Rate management cells	Counting lost packets
Event-driven (direct)	Handlers to react to changes in the network	Reacting to lost retransmissions

Figure 1: Examples for two dimensions of application/network coupling.

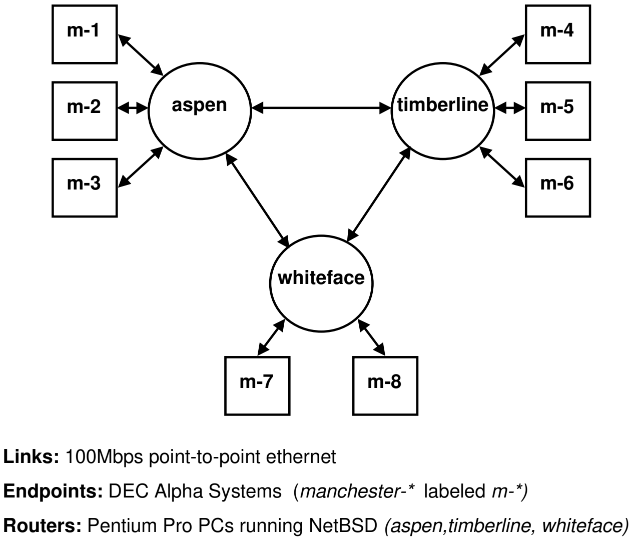

Figure 2: Testbed used for Airshed experiments

mance of the basic implementation with various adaptive versions.

The Airshed application [15] models formation, reaction, and transport of atmospheric pollutants and related chemical species. We implemented a distributed version of Airshed using Fx data parallelism [8]. Data parallelism in Fx is similar to High Performance Fortran [9], so these observations apply to other applications as well. An adaptive version of Airshed was developed using integrated task and data parallelism in Fx [24]. For efficient execution, this application involves significant communication in the form of array redistributions since the various chemistry and transport phases access the main particle array along different dimensions. The details of the Airshed implementation are described in [23].

We executed Airshed using a tool that automatically selects the best nodes for execution based on the network information provided by Remos. The details of this node selection procedure and its validation is discussed in [22]. Table1 presents the results obtained on our networking testbed, which consists of a number of DEC Alpha workstations connected via three routers. The testbed allows us to configure the bandwidth between the routers, as well as to apply various traffic patterns for controlled experiments. Figure 2 shows the set-up.

We observe that automatic node selection has little impact on performance in the absence of network traffic. In the presence of a fixed traffic stream that saturates one of the communication links, automatic node selection more than halves the execution time. The reason is that Airshed is a SPMD application with a significant communication component, and saturation of a single link creates a bottleneck that slows down the entire computation. However, it is possible to select a set of nodes automatically using Remos information such that the busy links are avoided for program communication. The last two columns in the table show performance on the network with load generators that simulate moderate utilization of network resources. We observe that the performance is considerably enhanced with automatic node selection as the node selection succeeds in avoiding congested links and busy processors in many cases. However, such enhancements are not always possible when the network is heavily used, and hence the performance advantage is not to the extent observed for a single congested link.

The results highlight the importance of simple adaptation in the *resource space* dimension at start-up time, and demonstrate that a toolset based on external measurements can effectively drive such adaptation. Note that internal measurements made by a program are not of any use in deciding which nodes the application should be started on. The main drawback of adaptation only at start-up time is that network conditions change over time, and hence adaptation during execution is important for long running applications.

Table2 presents preliminary results from a dynamic adaptive version of Airshed. This version queries Remos for network status after every major simulation step, and migrates to a new set of nodes if the current set of nodes or links become considerably more busy than other parts of the network. For the purpose of comparison, we created an adaptive version that would not actually migrate (but had adaptation support built into it) and compared it to the actual migrating adaptive application, under different network conditions. Both were started on the same set of nodes, and a fixed traffic pattern was maintained for the duration of

Execution Node Selection	Execution time with external load and traffic			
	No Network Traffic	Fixed Network Traffic	Dynamically varying Traffic	Dynamically varying Traffic and Load
Random	652	1726	1125	2121
Automatic	650	674	754	1420

Table 1: Performance results of 5-node Airshed in different network conditions. Execution times using automatic node selection are compared with those obtained with random node selection. For the case of dynamically varying traffic, only 1/4 of the Airshed processing was done in one invocation and the results shown are scaled up for comparison

the experiment. The interfering and non-interfering patterns are relative to the set of nodes and links on which the application was started.

We first observe that both the versions run slower than the non-adaptive versions of Airshed discussed earlier, even in the absence of any load or traffic. This observation points out the fixed overhead of adaptation support. In the absence of interfering network traffic, the migrating version executes slightly slower than the static version. This difference reflects two types of overheads associated with migration. First, there is a cost associated with analyzing and deciding the best nodes for execution. Second is the cost associated with unnecessary migration, which can happen because of the heuristic nature of adaptation decisions. Finally, we observe that the adaptive version performs significantly better in the presence of interfering traffic. The general conclusion is that support for adaptation entails moderate overheads, but it can minimize the impact of external traffic on execution times.

This experiment highlights the importance of runtime adaptation and demonstrates that an external tool like Remos is effective in driving the dynamic adaptation process. Note that runtime migration cannot be done with internal application measurements, since they are not available for the nodes that the application is not currently executing on. However, internal measurements can be used for load balancing, which is an example of an application modifying its demands in response to changes in the resource availability. It is clear that adaptation by migration exploits a degree of freedom in the resource dimension that is not available to load balancers.

5 Related work

An important contribution of our research is to provide a structure to adaptivity options, especially in the context of distributed simulations. We are not aware of any work that specifically addresses this aspect. However, a number of projects address measurement and management of network resources that complement the Remos system discussed in the paper. We also discuss some other approaches to adaptivity reported in the literature.

5.1 Network resource management and measurements

A number of resource management systems allow applications to make queries about the availability of computation resources, some examples being Condor [13] and LSF (Load Sharing Facility). Resource management systems for large scale internet-wide computing is an important area of current research, and some well known efforts are Globus [6] and Legion [7]. These systems provide support for a wide range of functions such as resource location and reservation, authentication, and remote process creation mechanisms. Recent systems that focus on measurements of communication resources across internet wide networks include Network Weather Service (NWS) [26] and topology-d [16]. NWS makes resource measurements to predict future resource availability, while topology-d computes the logical topology of a set of internet nodes. Both these systems actively send messages to make communication measurements between pairs of computation nodes. A number of sites are collecting Internet traffic statistics, e.g., [1]. This information is not in a 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 about the response times of different sites [21]. Related work also addresses estimating stochastic values [18] that represent varying quantities on networks.

In comparing with some of these projects, the Remos interface focuses on providing good abstractions and support for application level access to network status information and allows for a closer coupling of applications and networks. Remos implementations make measurements at network level when possible; this strategy minimizes the measurement overhead and yields key information for managing sharing of resources.

5.2 Other models and extensions

Several approaches to provide adaptivity without changes to the programming model have been researched in the lit-

Execution 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 2: Execution times of adaptive version of Airshed executing on a fixed set of nodes and on dynamically selected nodes

erature. Nevertheless, it is interesting to note that these systems include components that map directly into the concepts discussed in this paper.

A number of groups have looked at the benefits of explicit feedback to simplify and speed up adaptation (e.g., [10]). However, the interfaces developed by these efforts have been designed specifically for the scenarios being studied.

The *Quality Objects* QuO system[27] provides adaptivity in the context of object-oriented programming. To provide the feedback between applications and environment, the QuO system includes *system condition objects* that drive adaptivity either implicitly or explicitly.

An adaptive system that provides a shared-memory programming model for a network of workstations or PCs can take advantage of additional nodes and also deal with withdrawal of a nodes [17]. Here the control of adaptivity rests with the (software) distributed shared-memory system, but the application (or the compiler) determines the points in the execution of the program where adaptivity is possible.

6 Concluding remarks

Figure 1 gives examples of the 4 different kinds of couplings between applications and networks that are discussed in this paper. Network-aware applications today focus overwhelmingly on implicit interaction. We argue that this state of affairs is due to the current (lack of) support for other interactions models by network architectures. As network architectures begin to provide information that is more accurate, more timely, and more detailed, network-aware applications will be motivated to also explore explicit interaction.

We show how the adaptivity options for distributed simulations fit in the general framework of adaptive execution. The results demonstrate that portable external mechanisms for network measurements are necessary to support effective adaptive execution of large distributed scientific applications.

Acknowledgments

We appreciate contributions and comments by D. Bakken, J. Bolliger, P. Dinda, B. Lowekamp, D. O'Hallaron, N. Miller, D. Sutherland, and J. Zinky.

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.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] http://www.nlanr.net.

[2] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Trans. Softw. Eng.*, 24(5):376 – 390, May 1998.

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

[4] Dave Clark, S. Shenker, and L. Zhang. 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*, pages 14–26, Baltimore, August 1992. ACM.

[5] W. Eddy, M. Fitzgerald, C. Genovese, A. Mockus, and D. Noll. Functional image analysis software - computational olio. In A. Prat, editor, *Proceedings in Computational Statistics*, pages 39–49, Heidelberg, 1996.

[6] I. Foster and K. Kesselman. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications*, 11(2):115–128, 1997.

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

[8] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology*, 2(3):16–26, Fall 1994.

[9] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, December 1996.

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

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

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

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

[14] Bruce Lowekamp, Nancy Miller, Dean Sutherland, Thomas Gross, Peter Steenkiste, and Jaspal Subhlok. A Resource Query Interface for Network-Aware Applications. In *7th IEEE Symposium on High-Performance Distributed Computing*, pages 189–196, Chicago, July 1998.

[15] G. McRae, A. Russell, and R. Harley. *CIT Photochemical Airshed Model - Systems Manual*. Carnegie Mellon University, Pittsburgh, PA, and California Institute of Technology, Pasadena, CA, February 1992.

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

[17] A. Scherer, H. Lu, T. Gross, and W. Zwaenepoel. Transparent adaptive parallelism on nows using openmp. In *Proc. 7thACM Symp. on Principles and Practice of Parallel Prog. (PPoPP'99)*, page (to appear), Atlanta, GA, May 1999. ACM.

[18] J. Schopf and F. Berman. Performance prediction in production environments. In *12th International Parallel Processing Symposium*, pages 647–653, Orlando, FL, April 1998.

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

[20] Peter Steenkiste. Adaptation models for network-aware distributed computations. In *3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99)*, Orlando, January 1999. IEEE. Springer-Verlag.

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

[22] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[23] J. Subhlok, P. Steenkiste, J. Stichnoth, and P. Lieu. Airshed pollution modeling: A case study in application development in an HPF environment. In *12th International Parallel Processing Symposium*, pages 701–710, Orlando, FL, April 1998.

[24] J. Subhlok and B. Yang. 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*. ACM, June 1997.

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

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

[27] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997.

7 Biographies

Thomas R. Gross is a faculty member in the School of Computer Science at Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA. (thomas.gross@cs.cmu.edu). He joined CMU in 1984 after receiving a Ph.D. in Electrical Engineering from Stanford University. He also has an appointment at ETH Zurich. He is interested in tools, techniques, and abstractions for software construction and has worked on many aspects of the design and implementation of programs. To add some realism to his research, he has focussed on compilers for uni-processors and parallel systems. He has worked on many areas of compilation (code generation, optimization, debugging, partitioning of computations, data parallelism and task parallelism) and software construction (frameworks, patterns, components). In his current research, Thomas Gross and his colleagues investigate network- and system-aware programs – i.e. programs that can adjust their resource demands in response to resource availability.

Further information (including additional references and downloadable versions of many papers) can be found at www.cs.cmu.edu/ cmcl

Peter Steenkiste received the degree of Electrical Engineer from the University of Gent in Belgium in 1982, and the MS and PhD degrees in Electrical Engineering from Stanford University in 1983 and 1987, respectively. He then joined the School of Computer Science at Carnegie Mellon University, where he is current a Senior Research Scientist.

Peter Steenkiste's research interests are in the areas of networking and distributed computing. While at CMU, Peter Steenkiste worked on a number of projects in the high-performance networking and distributed computing area. Earlier projects include Nectar, the first workstation clusters built around a high-performance, switch-based local area network, Gigabit Nectar, a heterogeneous multi-computer, and Credit Net, a high-speed ATM network. Peter Steenkiste is currently exploring the notion of "application-aware networks", i.e. networks that can delivery high quality, customized services to applications, in the context of the Darwin project.

He is also involved in the Remulac project, which is developing middleware in support of network-aware applications. More information can be found on his web page http://www.cs.cmu.edu/ prs

Peter Steenkiste is a member of the IEEE Computer Society and the ACM. He has been on a number of program committees and is an associated editor for IEEE Transactions on Parallel and Distributed Systems.

Jaspal Subhlok received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India in 1984, and Ph.D. degree in computer science from Rice University, Houston in 1990. Between 1990 and 1998 he served as a member of the research faculty in the School of Computer Science at Carnegie Mellon University, Pittsburgh. He is currently an Associate Professor of Computer Science at the University of Houston.

Dr Subhlok's technical areas of interest are compilers, tools and runtime systems, particularly in the context of parallel and distributed computing. His research involves design of algorithms and systems to solve a variety of problems in programming and runtime support for parallel and networked systems. The focus of his current research is on "network aware" distributed computing, that spans the development of tools and frameworks to support applications that can dynamically adapt to changing system resources. His earlier projects included development and standardization of integrated task and data parallelism in the context of High Performance Fortran, algorithms and tools for automatic mapping of mixed task and data parallel programs, and validation of job scheduling strategies with actual supercomputer workloads.

Further information is available from http://www.cs.uh.edu/ jaspal.