# Multimethod Communication for High-Performance Metacomputing Applications

Ian Foster† Jonathan Geisler† Carl Kesselman‡ Steve Tuecke†

†Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
U.S.A.
{foster,geisler,tuecke}@mcs.anl.gov

‡Beckman Institute
California Institute of Technology
Pasadena, CA 91125
U.S.A.
carl@compbio.caltech.edu

Preprint ANL/MCS-P601-0596

**Abstract**

Metacomputing systems use high-speed networks to connect supercomputers, mass storage systems, scientific instruments, and display devices with the objective of enabling parallel applications to access geographically distributed computing resources. However, experience shows that high performance often can be achieved only if applications can integrate diverse communication substrates, transport mechanisms, and protocols, chosen according to where communication is directed, what is communicated, or when communication is performed. In this article, we describe a software architecture that addresses this requirement. This architecture allows multiple communication methods to be supported transparently in a single application, with either automatic or user-specified selection criteria guiding the methods used for each communication. We describe an implementation of this architecture, based on the Nexus communication library, and use this implementation to evaluate performance issues. The implementation supported a wide variety of applications in the I-WAY metacomputing experiment at Supercomputing 95; we use one of these applications to provide a quantitative demonstration of the advantages of multimethod communication in a heterogeneous networked environment.

## 1 Introduction

Future networked computing systems will be increasingly heterogeneous in terms of both the types of networked devices and the capabilities of the networks used to connect these devices. At the same time, the applications that run on these networks are becoming more sophisticated

in terms of the computations they perform and the types of data that they communicate [6]. The various gigabit testbeds showcased early examples of high-performance networked applications, while in the I-WAY networking experiment at Supercomputing 95, around sixty groups demonstrated applications designed to exploit networked supercomputers, mass storage systems, scientific instruments, and advanced display devices [10].

Experiences on the I-WAY and other networking testbeds show that metacomputing applications often need to exploit multiple network interfaces, low-level protocols, data encodings, and quality of service choices if they are to achieve acceptable performance. Coupled models, which use multiple supercomputers to exploit large aggregate memory or to run different components more quickly on different architectures, need to use machine-specific communication methods within computers and optimized wide area protocols between computers [21, 22]. Collaborative environments require a mixture of protocols providing different combinations of high throughput, multicast, and high reliability [11, 12]. Applications that connect scientific instruments or other data sources to remote computing capabilities need to be able to switch among alternative communication substrates in the event of error or high load [20]. In general, the choice of communication method can vary according to where communication is directed, what is communicated, or when communication is performed.

Metacomputing applications requiring multiple communication methods have previously been developed in an ad hoc fashion, with different program components coded to use different low-level communication mechanisms. While effective, this approach is tedious, error prone, and nonportable. A simpler approach would be to allow programmers to develop applications using a single high-level notation, such as the Message Passing Interface (MPI) [19] or a parallel language, and then provide mechanisms that allow the methods used for each communication to be determined independently of the program text. However, the realization of this approach requires solutions to challenging problems: *separate specification* of communication operation and communication method; *identification* of applicable communication methods; *selection* from among alternative methods; and the *incorporation* of multiple communication methods into an implementation.

In this article, we describe a software architecture that addresses the problems just listed. This architecture allows programmers to specify communications in terms of high-level abstractions such as message passing or remote procedure call, while supporting diverse low-level methods for actual communications. Communication methods can be associated with individual communication operations, and the selection of an appropriate method can be guided by both automatic and user-specified criteria. The architecture also incorporates solutions to various problems that arise when multiple communication methods are incorporated into a single implementation. Central to this multimethod communication architecture is an abstraction called a communication link, which provides a concise, mobile representation of both the target of a communication operation and the methods used to perform that operation.

These multimethod communication techniques have been implemented in the context of the Nexus multithreaded runtime system [15, 16]. Nexus has been used to implement a variety of parallel languages and communication libraries [7, 14, 12], including the MPI implementation used extensively in the I-WAY wide area computing experiment [10]. We use Nexus to study the performance of alternative approaches to the implementation of various multimethod communication structures. We conclude with a case study in which our multimethod communication techniques are used to improve dramatically the performance of an MPI-based climate model.

In brief, the contributions of this article are as follows:

1. The definition of a software architecture that permits application-level communications to

be specified independently of the low-level methods used to perform communication and that supports both automatic and manual selection of the methods used for particular communication operations.

2. The description and evaluation of implementation techniques that support the simultaneous use of multiple communication methods within a single application.

3. A demonstration that multimethod communication can significantly improve the performance of realistic scientific applications.

# 2 Multimethod Communication

The need for multiple communication methods in a single application can arise for a number of reasons, some of which we consider here.

- *Transport mechanisms.* Complex applications such as those demonstrated on the I-WAY may integrate diverse computational resources, including visualization engines, parallel supercomputers, and database computers [10, 20, 21, 22]. While the Internet Protocol (IP) provides a standard transport mechanism for routed networks [8], parallel computers and local area networks often support alternative, more efficient mechanisms. As we will show in Section 4, the use of specialized transport mechanisms can be crucial to application performance.

- *Network protocols.* Many network services are available in addition to the point-to-point reliable delivery typically provided by message-passing libraries. Applications such as collaborative engineering [12] can exploit specialized protocols such as Unreliable Datagram Protocol (UDP), IP multicast, reliable multicast, and Realtime Transport Protocol (RTP) or application-specific protocols for selected data, such as shared state updates and video.

- *Quality of service (QoS).* Future networks will support channel-based QoS reservation and negotiation [26, 4]. High-performance multimedia applications probably will want to reserve several channels providing different QoS; for example, they might use a low-latency, low-bandwidth channel for control information and a high-bandwidth, unreliable channel for image data transfer.

- *Interoperability of tools.* Parallel applications must increasingly interoperate with other communication paradigms, such as CORBA and DCE. In heterogeneous environments, an MPI program may need to interoperate with other MPI implementations. In each case, different protocols must be used to communicate with different processes.

- *Security.* Different mechanisms may be used to authenticate or protect the integrity or confidentiality of communicated data [27], depending on where communication is directed and what is communicated. For example, control information might be encrypted outside a site, but not within, while data is not encrypted in either case.

These examples show that it can be necessary to vary the methods used for a particular communication according to *where* communication is directed, *what* is communicated, and even—since many of the choices listed above can vary over time—*when* communication is performed.

## 2.1 Requirements

Recognizing that multimethod communication is important, we face the challenge of developing tools and techniques that allow programmers to use multiple communication methods efficiently without introducing overwhelming complexity. We argue that a fundamental requirement is that the programmer be able to specify communications in terms of a single abstraction (whether message passing, remote procedure call, etc.), independently of the low-level method used to effect a particular communication. In addition, it should be easy to distinguish communications intended for a particular purpose (for example, communications directed to a particular remote location), so that programmers can associate different methods with different subsets of the communication operations within a program. The examples presented above show that it is not enough to specify communication method based solely on the source and destination processors.

Implementations of multimethod communication must permit the coexistence of multiple methods within a single application. This is a nontrivial problem, since different methods may use quite different mechanisms for initiating and processing communications. It is also important to have flexible techniques for selecting the communication method to be used. While ease of use demands automatic selection mechanisms, programmer-directed selection must also be supported, and automatic and programmer-directed selection must be able to coexist. For example, automatic selection might be used to determine whether to use shared memory or TCP/IP between two computers, while manual selection could be used to specify that data is to be compressed before communication. For some communication methods, programmers need to manage low-level behavior by specifying values for important parameters. For example, a TCP-based method might allow a programmer to specify socket buffer sizes.

Finally, both automatic and manual selection require access to information about the availability and applicability of different communication methods and about system state and configuration. For example, shared-memory communication is appropriate only if directed to another process within the same shared address space. An implementation of multimethod communication must provide this information via enquiry functions. Enquiry functions should also enable programmers to evaluate the effectiveness of automatic selection or to tune manual selections.

## 2.2 Communication Primitives

The preceding discussion has identified requirements for an implementation of multimethod communication. These requirements can be satisfied in a variety of ways. We advocate an approach based on a one-sided asynchronous communication mechanism implemented by a *communication link* and *remote service request*.

Before going into details of our approach, let us consider the limitations of supporting multimethod communication with traditional two-sided message passing primitives. We start with the observation that two-sided communication defines a specific protocol for synchronizing and extracting data at the receive side of the transfer. This protocol can hinder communication methods, such as stream communication, in which an explicit receive operation may not be appropriate.

Message-passing libraries such as PVM [17], MPL, or NX provide no notion of communication context: a receive can potentially match any send. This feature makes it difficult to associate a communication method with a specific set of communication operations or to support different methods on different communication operations. The situation is improved in MPI [19] by the introduction of communicators, which provide a scope for communication.

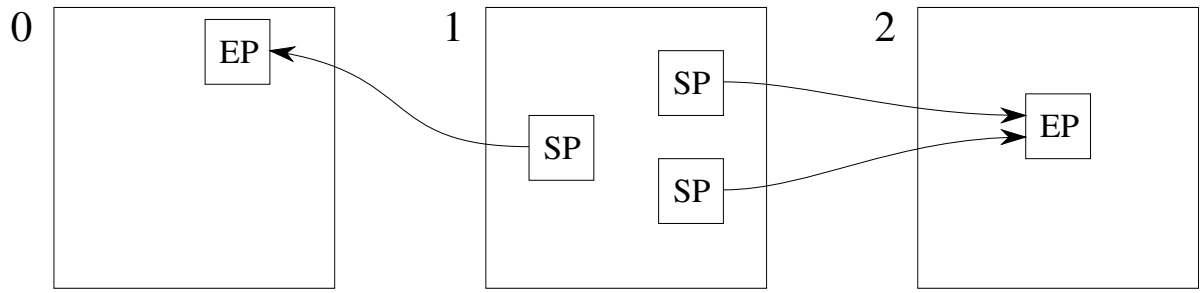


Figure 1: The communication link and its role in communication. The figure shows three address spaces; three startpoints in address space 1, labeled SP, are linked to endpoints labeled EP in address spaces 0 and 2.

One could associate a communication method with a communicator. However, communicators also have their limitations. A communicator defines a symmetric communication structure and must be created by a collective operation. This symmetry may cause difficulties in the case of asymmetric communication methods, such as multicast. Additionally, program design is complicated in that the communicators used for different methods must be managed explicitly. Also, as communicators cannot be transferred between nodes, it is difficult for one node to inform another node of its communication preference.

In light of these limitations, we choose to support multimethod communication with communication links and remote service requests rather than point-to-point message passing. In our system, communication is directed over a communication link that connects a *communication startpoint* to a *communication endpoint*. (We have adopted this terminology in preference to the term *global pointer* used in prior publications [15, 16], because the latter led many readers to assume a distributed shared memory.) Before a startpoint can be used, it must be explicitly bound to an endpoint to form a communication link. If more than one startpoint is bound to an endpoint, incoming communications are merged, just as incoming messages to the same node are merged in a point-to-point message passing system. If a startpoint is bound to more than one endpoint, communication results in a multicast operation. Startpoints can be copied between processors, but endpoints cannot. When a startpoint is copied, new communication links are created, mirroring the links associated with the original startpoint. This ability to copy a startpoint means that startpoints can be used as global names for objects. These names can be used anywhere in a distributed system.

A communication link supports a single communication operation: an asynchronous *remote service request* (RSR). An RSR is applied to a startpoint by providing a procedure name and a data buffer. For each endpoint linked to the startpoint, the RSR transfers the data buffer to the address space in which the endpoint is located and remotely invokes the specified procedure, providing the endpoint and the data buffer as arguments. A local address can be associated with an endpoint, in which case any startpoint associated with the endpoint can be thought of as a "global pointers" to that address.

Each communication link defines a unique, asymmetric communication space with which a specific communication method can be associated. A process can create any number of links, allowing communications intended for different purposes to be distinguished (see Figure 1). Since the communication method is associated with the communication link, no additional

bookkeeping needs to be performed by the application.

A key to the utility of the communication link abstraction is the portability of the startpoint. A process can create a link, associate a communication method with the startpoint, and then communicate that startpoint to other processes, providing those processes with a handle that they can use to perform RSRs to the remote location. In addition, the communication method associated with any startpoint can be altered, so a process receiving a startpoint can change the communication method to be used, on the basis of sender-side requirements. In general, then, the communication link and RSR abstractions overcome the limitations of two-sided communication primitives for multimethod communication.

# 3 Implementing Multimethod Communication

We now turn our attention to the techniques used to implement multimethod communication. We describe these techniques in the context of Nexus [15, 16], a portable, multithreaded communication library designed for use by parallel language compilers and higher-level communication libraries. In the discussion that follows, we refer to an address space, or virtual processor, as a *context.* In the communication architecture that we present below, the range of possible communication methods available to a computation is defined by the contexts in which the startpoint and endpoint reside; the actual method used for a particular RSR is determined by the data structures associated with the startpoint and endpoint.

## 3.1 Multimethod Communication Architecture

Figure 2 provides an overview of the data structures used to support multiple communication methods. A communication method is implemented by a *communication module.* Each communication module implements a standard interface that includes communication-oriented functions, an initialization function, and functions used to construct communication descriptors and communication objects. To enable the coexistence of many different communication modules within an executable, Nexus accesses interface functions within a module via a *function table,* constructed when the module is loaded. To date, communication modules have been constructed for local (intracontext) communication, TCP sockets, Intel NX message passing, IBM MPL, AAL-5 (ATM Adaptation Layer 5), Myrinet, unreliable UDP, and shared memory; others are being developed.

Several methods are provided for determining which communication modules can be used by a particular executable. When the Nexus library is built, a default set of modules is defined. Additional communication modules can be specified by entries in a resource database, by command line arguments, or by function calls from within the program. The function table interface is designed so that if a required module has not been compiled into the Nexus library, it can be loaded dynamically.

A *communication descriptor* contains the information that a communication module needs in order to communicate with a specific context. For example, when using MPL to communicate between nodes on IBM SP multicomputers, a communication descriptor contains a node number and a globally unique session identifier, which is used to distinguish between different SP partitions. On the Intel Paragon, the descriptor also includes the name of the process with which we wish to communicate, since on the Paragon, a parallel computation can contain several processes executing on the same processor. Communication descriptors are grouped into

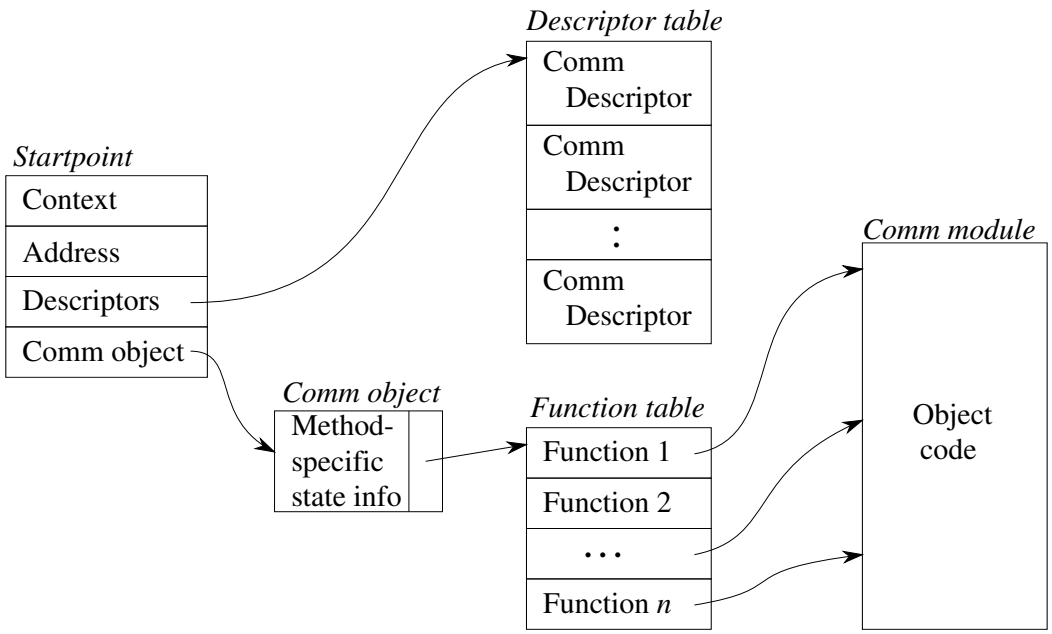


Figure 2: Nexus data structures used to support multimethod communication. The figure shows the startpoint, communication descriptor table, communication object, function table, and communication module. The data structures are explained in the text.

a *communication descriptor table*, which is a concise and easily communicated representation of information about communication methods.

An active connection is represented by a *communication object*. A communication object contains the information found in a single communication descriptor, a pointer to the function table corresponding to that descriptor, and any additional state information needed to represent the connection. For example, a communication object for a TCP connection contains the file descriptor for the TCP socket.

Finally, a *startpoint* contains context and address information for the link's endpoint, a pointer to the communication descriptor table for the context being referenced, and a communication object representing the communication method currently being used by the pointer. Communication objects are shared among startpoints that reference the same context and use the same communication method.

When a startpoint is created in a context, the communication descriptor table representing the methods supported by that context is attached to the startpoint along with a communication object referencing the "local" communication method. When the startpoint is transferred to another context, this descriptor table is passed with it. Hence, any context receiving a startpoint also receives the information required to communicate to the referenced endpoint. Before a startpoint can be used, one of the methods specified in its descriptor table must be selected and that method used to construct a communication object. Subsequent operations on the startpoint then occur via the communication object. The techniques used to select a communication method are discussed below. Note that this technique allows us to change dynamically the communication method used by a specific startpoint, simply by constructing a new communication object and storing a reference to that object in the startpoint.

The mechanisms that we have described are very general and hence powerful, but make startpoints rather heavyweight entities. While this situation is acceptable in a wide-area context, where the cost of communicating a few tens of bytes of descriptor table is insignificant, it can be unacceptable in more tightly coupled systems. Fortunately, it is possible to recognize special cases in which a default descriptor table is used repeatedly, as is often the case with communication links between nodes within a parallel computer. In such situations, the size of a startpoint and the cost of manipulating it can be reduced significantly by not attaching a descriptor table.

## 3.2 Selecting a Communication Method

Upon receipt of a startpoint, a context must determine which of the methods contained in the attached descriptor table are to be used for subsequent communication using that startpoint. As explained in Section 2.1, we wish to support both automatic and manual method selection.

Nexus currently uses a simple automatic selection rule: a received descriptor table is scanned in order and the first "applicable" communication method is used. A method is "applicable" if it is supported by both the local and remote contexts and meets additional method-specific criteria. For example, the MPL communication method can be used only if both contexts reside in the same SP partition.

Figure 3 illustrates automatic method selection. Consider a network configuration in which three nodes are connected by an Ethernet. Nodes 1 and 2 are part of an IBM SP2 and hence are also connected by MPL. Node 0 has a communication link to node 2. Because the associated startpoint was received from node 2, its attached descriptor table contains entries for both Ethernet (E) and MPL (M). However, node 0 supports only Ethernet and so this method is used. The startpoint is then migrated to node 1. On arrival at node 1, we determine that MPL is applicable, since it is supported by both nodes and since both nodes are on the same SP partition.

Because of the ordered scan of the descriptor table, placing the MPL descriptor before the Ethernet descriptor results in a "fastest first" selection policy. This policy is easily extended. For example, network QoS parameters be incorporated into the selection policy, by looking at available network bandwidth rather than raw bandwidth before indicating that a module is acceptable. The user can also influence the choice of method by reordering entries within the communication descriptor table or by adding or deleting descriptors.

## 3.3 Detecting and Processing Multimethod Communication

Startpoints represent the sending side of a communication link; at the endpoint, incoming RSRs must be detected and processed. Since different communication methods may require different detection mechanisms, we must consider how incoming communication can be identified across all the communication methods available to a context.

A straightforward approach to checking for pending communication is to provide a single polling function that iterates over the elements of a context's communication descriptor table, invoking a method-specific poll operation for each entry. To evaluate the performance of this approach, we conducted experiments with a ping-pong microbenchmark that bounces a vector of fixed size back and forth between two processors a large number of times. This process is repeated to obtain one-way communication times for a variety of message sizes.

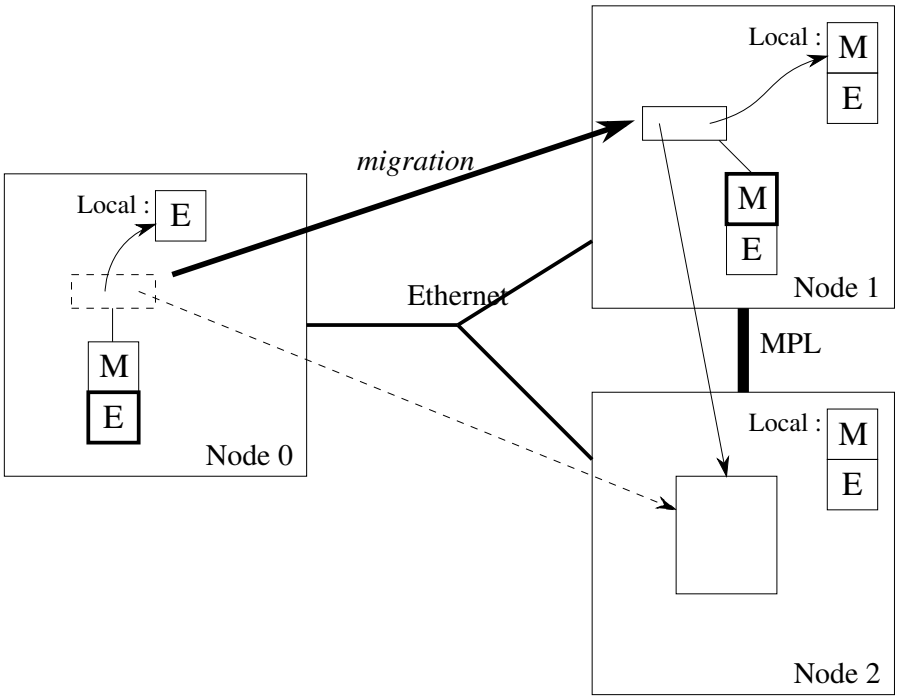


Figure 3: Communication method selection in Nexus. See text for details.

We measured performance of three implementations of the ping-pong microbenchmark: a pure MPL version, a Nexus version supporting a single communication method (MPL), and a Nexus version supporting two communication methods (MPL and TCP). In both Nexus versions, all communications were initiated with MPL; hence, any performance degradation in the MPL/TCP Nexus version is due to overhead associated with TCP polling. All experiments were run on the IBM SP2 at Argonne National Laboratory, which consists of Power 1 processors connected via an SP2 multistage switch. Both MPL and TCP operate over the switch and can achieve maximum bandwidths of about 36 and 8 MB/sec, respectively.

Figure 4 shows our results. The lower two lines in the first graph show that for small messages, the message-driven execution model supported by Nexus introduces some overhead on the SP2, relative to native MPL; we have provided a detailed analysis of these overheads elsewhere [16]. In the other graph, these same two lines coincide, thus indicating that Nexus overheads are not significant for larger messages.

The upper two lines in each graph reveal a disadvantage of the unified polling scheme. In general, the cost of a poll operation can vary significantly depending on the mechanism used. For example, on many parallel computers, the probe operation used to detect communication from another processor is cheap, while a TCP `select` is expensive. On the SP2, the `mpc_status` call used to detect an incoming MPL operation costs 15 microseconds, while a `select` costs over 100 microseconds. A consequence of this cost differential is that an infrequently used, expensive method imposes significant overhead on a frequently used, inexpensive method. On the SP2, the cost for a zero-byte message as measured by the ping-pong microbenchmark increases from 83 to 156 microseconds with TCP polling, even though no TCP communication is performed. In addition, TCP support degrades MPL communication performance even for large messages.

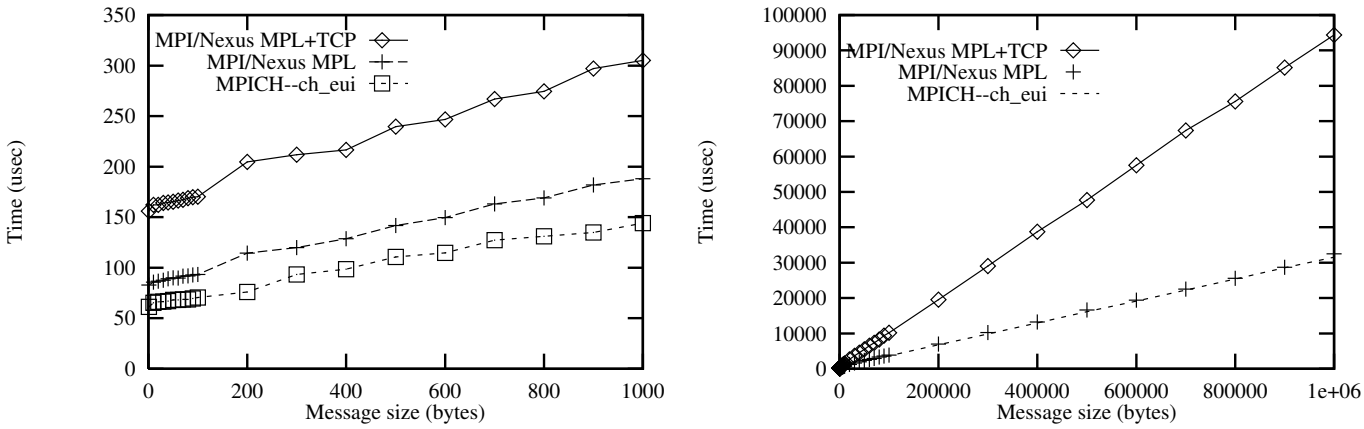


Figure 4: One-way communication time as a function of message size, as measured with both a low-level MPL program and the ping-pong microbenchmark, using single-method and multimethod versions of Nexus. On the left, we show data for message sizes in the range 0–1000, and on the right a wider range of sizes. See the text for details.

We hypothesize that this degradation is because repeated kernel calls due to `select` slow the transfer of data from the SP2 communication device to user space.

To reduce the additional overhead incurred in detecting multimethod communication, we take advantage of the fact that the high latency inherent in the TCP interface means that TCP messages will be delivered less frequently than MPL messages. Hence, it is acceptable to check for TCP communications less frequently than MPL communications. To apply this optimization, we extend Nexus to support a parameter, `skip_poll`, that specifies the frequency with which TCP polls should be performed. For example, with a `skip_poll` value of 2, the TCP interface will be checked every other time the polling function is called, while the MPL interface will be checked every time. Note that the overall frequency at which the polling function will be called will depend on characteristics of both the application program and the Nexus implementation. However, the polling function will be called at least every time a Nexus operation is performed.

To demonstrate the effectiveness of the `skip_poll` parameter, we use a second microbenchmark that runs two instances of the ping-pong program concurrently, one over MPL and the second over TCP (Figure 5). The two programs execute until the MPL ping-pong has performed a fixed number of roundtrips. Then the one-way communication time of each pair is computed. To simulate an environment in which we have two separate SP2s coupled by a high speed network, we place the endpoints for the TCP communication in separate partitions, a software abstraction provided on the SP2. Processors in the same partition can communicate by using either TCP or IBM's proprietary Message Passing Library (MPL), while processors in different partitions can communicate via TCP only.

The experiment was repeated for a range of `skip_poll` values, yielding the results shown in Figure 6. The performance of the MPL ping-pong is degraded significantly by the concurrently executing TCP ping-pong. As we might expect, MPL performance improves with increasing `skip_poll`, while TCP performance degrades. For this experiment, we can see that a `skip_poll`

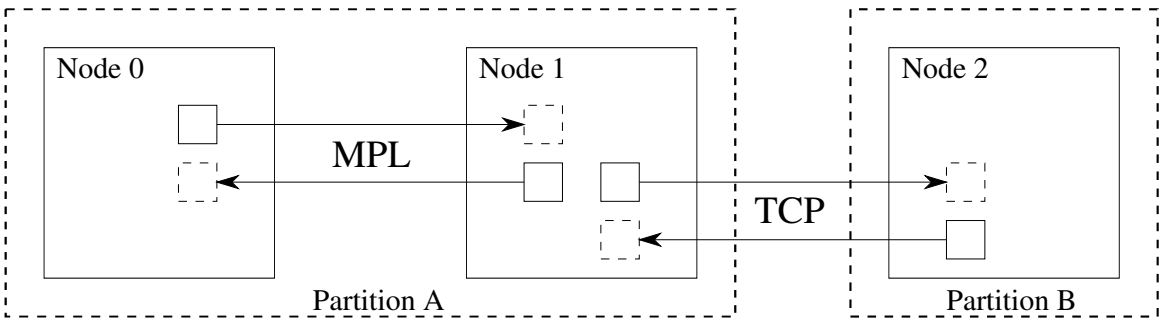


Figure 5: Configuration for the multiple ping-pong communication benchmark. Two ping-pong programs run concurrently, one within an IBM SP2 partition using MPL, and the other between two partitions using TCP.

values of around 20 provides improvement in MPL performance, while not impacting TCP performance significantly for both small and large message sizes.

The polling function can be refined further on systems that allow a separate thread of control to block awaiting communication. (IBM's AIX 4.1 operating system provides this capability for TCP communication but is not installed on the Argonne SP2.) On such systems, we can create a specialized polling function that executes in its own thread of control and checks only those communication methods for which blocking is supported. Preliminary experiments show that this approach allows TCP communication operations to be detected without significant impact on MPL performance.

Another approach to the problem of processing incoming communications associated with multiple communication methods is to define a dedicated *forwarding* processor. This processor receives all incoming communication associated with a specific communication method and forwards these communications to their intended destination by using an alternative method. For example, in an SP2 environment, all TCP communications from external sources would be routed to a single SP node, which in turn would forward these communication to other nodes by using MPL. The use of a forwarding node means that other nodes need not check for communications with the forwarded communication method. We have implemented TCP forwarding in Nexus; performance results are presented in the next section.

# 4 Case Study: A Coupled Climate Model

The multimethod communication techniques described in this article were used in the I-WAY networking experiment to support a variety of applications, ranging from coupled simulations to collaborative environments and networked instruments [10, 20]. In this section, we consider one such application—a coupled simulation—and study its behavior in a controlled environment similar to the I-WAY wide area network.

In general, scientific simulations synchronize too frequently to permit distributed execution. One exception is multicomponent models constructed by coupling models of distinct subsystems [21, 22]. In such models, communication and synchronization between submodels is often less frequent than internal communication and synchronization. Hence, it can be feasible to execute distinct components on different supercomputers. This distribution can have advantages.

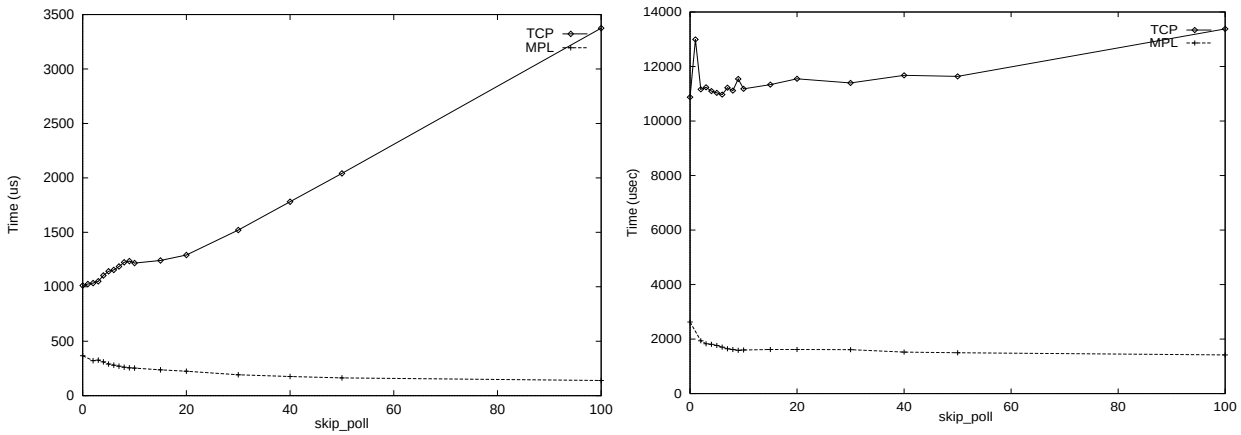


Figure 6: One-way communication time as a function of `skip_poll` for a microbenchmark in which two ping-pong programs run concurrently over MPL and TCP, as described in the text. The graph on the left is for zero-length messages, and the graph on the right is for 10 kilobyte messages.

In some cases, different models may execute more efficiently on one computer than another; for example, one component may execute more quickly on a vector supercomputer, while another is better suited to an parallel computer [21]. In other cases, distributed execution can provide access to larger aggregate memory and hence permit the solution of larger problems [22].

The model that we study here is the Millenia coupled climate model, designed to run at relatively low resolutions for multicentury simulations. This model uses MPI for communication and combines a large atmosphere model (the Parallel Community Climate Model [13]) with an ocean model (from U. Wisconsin). The two models execute concurrently and perform considerable internal communication. Every two atmosphere steps, the models exchange information such as sea surface temperature and various fluxes. The models typically run for tens or hundreds of thousands of timesteps.

To provide a controlled environment for our experiments, we run the two model components not on two different computers but instead on distinct partitions of the Argonne SP2. As noted above, the SP2 programming environment permits the use of the fast MPL communication library only within a partition; communication between partitions must be performed with TCP (Figure 7). Since TCP over the SP2 switch runs at about 8 MB/sec and incurs small-message latencies of around 2 milliseconds, this two-partition configuration has similar performance characteristics to two SP2 systems connected by a tuned OC3 or faster ATM network in a metropolitan area network. In our experiments, the atmosphere model runs on 16 processors and the ocean model on 8 processors. Communication is achieved by using the MPICH [18] implementation of MPI layered on top of Nexus. This layering adds an execution time overhead of about 6 percent when compared with MPICH running on top of MPL.

We measured execution times for the coupled model both without multimethod communication and with various multimethod communication techniques. In the absence of multimethod communication support, both interpartition and intrapartition communication must be performed with TCP. This requirement results in a total execution time an order of magnitude greater than the worst multimethod time, clearly demonstrating the advantages of multimethod

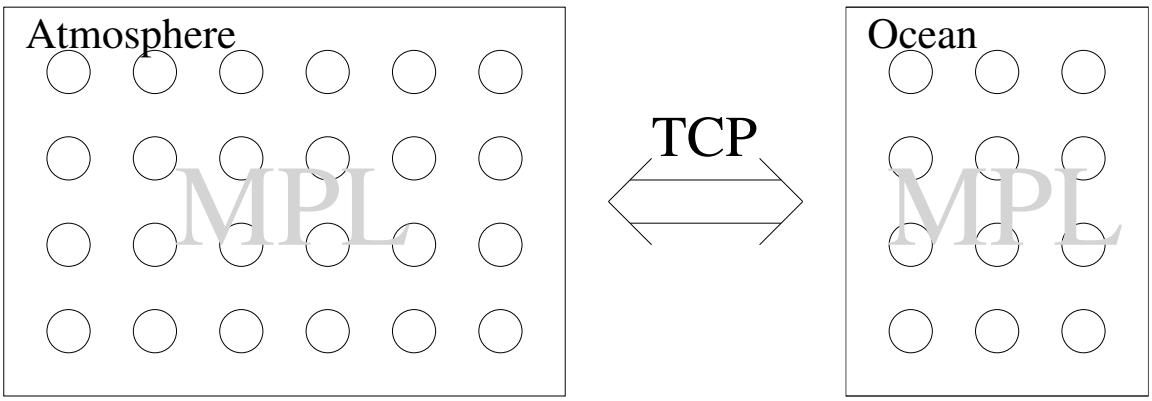


Figure 7: The Argonne/Wisconsin coupled ocean/atmosphere model in the configuration used for our multimethod communication experiments, showing the two IBM SP partitions.

Table 1: Time spent in communication between models and total execution time for the coupled model. Times are in seconds per timestep on 24 processors.

| No. | Experiment | Total |
|---|---|---|
| 1 | Selective TCP | 104.9 |
| 2 | Forwarding | 109.3 |
| 3 | skip poll 1 | 109.1 |
| 4 | skip poll 100 | 107.8 |
| 5 | skip poll 10000 | 105.4 |
| 6 | skip poll 12000 | 105.0 |
| 7 | skip poll 13000 | 108.3 |

communication.

Table 1 shows the execution time per timestep when using various multimethod communication techniques. Row 1 shows a best-case scenario (given for comparative purposes), in which TCP polling is enabled only in the section of code in which partitions communicate. Row 2 shows the performance of multimethod communication using TCP forwarding, while rows 3 through 7 show the execution time for various `skip_poll` values. As we can see, the performance of the `skip_poll` implementation increases from values of 0 to 12,000 and then degrades as the decreased TCP polling frequency increases the latency of intermodel communication. As can be seen from row 6, the use of a `skip_poll` value of 12,000 results in performance that is within 0.1 percent of the best case result.

Note that the performance of the polling implementation can exceed that of TCP forwarding. This result can be explained by the fact that nodes on the SP have good TCP connectivity and the use of a forwarder incurs additional overhead not found in the polling implementation.

# 5 Related Work

Many researchers have proposed and investigated communication mechanisms for heterogeneous computing systems (for example, [1, 3, 23]). However, this work has typically been concerned with hiding heterogeneity by providing a uniform user-level interface rather than with exploiting and exposing the heterogeneous nature of networks and applications.

Some communication libraries permit different communication methods to coexist. For example, p4 and PVM on the Intel Paragon use the NX communication library for internal communication and TCP for external communication [5, 17]; p4 supports NX and TCP within a single process, while PVM uses a forwarding process for TCP. In both systems, the choice of method is hard coded and cannot be extended or changed without substantial re-engineering.

The x-kernel [24] and the Horus distributed systems toolkit [30] both support the concurrent use of different communication methods. Horus provides some support for varying the communication method associated with an entire group. However, it does not provide for automatic method selection or for the migration of communication capabilities (with associated method information) between processes. In other respects, the x-kernel and Horus complement our work by defining a framework that supports the construction of new protocols by the composition of simpler protocol elements. These mechanisms could be used within Nexus to simplify the development of new communication modules. Early results with Horus suggest that these compositional formulations simplify implementation but can introduce overheads similar to those encountered when layering MPICH on Nexus: additional message header information, function calls, and messages. Tschudin [28] and the Fox project [2] have explored similar concepts and report similar results.

Finally, we note that concepts similar to the Nexus communication link are used in other systems. For example, Split-C [9] uses a global pointer construct to support remote put and get operations within homogeneous systems. Nexus mechanisms also share similarities with Active Messages [29] and Fast Messages [25]. However the association of communication method choices with startpoints is unique to Nexus.

# 6 Conclusions

We have described techniques for representing and implementing multimethod communication in heterogeneous environments. We use a startpoint construct to maintain information about the methods that can be used to perform communications directed to a particular remote location. Simple protocols allow this information to be propagated from one node to another and provide a framework that supports both automatic and manual selection from among available communication methods. These techniques have been incorporated in the Nexus communication library and used to support a wide variety of metacomputing applications in the I-WAY wide-area computing experiment.

We have used the Nexus runtime system to illustrate the implementation of the various techniques described in this article. Performance studies using both microbenchmarks and a coupled climate model application provide insights into the costs associated with multimethod communication mechanisms. In particular, we show that careful management of polling frequencies can improve multimethod communication performance significantly and can provide performance superior to techniques based on a forwarding processor.

The results reported in this article suggest several directions for future work. Polling func-

tions can be further refined, for example to allow for adaptive adjustment of `skip_poll` values and the use of blocking operations. The basic framework can be extended to support additional communication methods. Streaming protocols, security-enhanced protocols, and multicast are currently being investigated; while preliminary design work suggests that they fit the framework well, practical experience may suggest refinements. We also plan to investigate more sophisticated heuristics for automatic method selection. Further work is also required on the representation, discovery, and use of configuration data, particularly in situations where it is subject to change.

# Acknowledgments

Our understanding of multimethod communication issues has benefited greatly from discussions with Steve Schwab, who prototyped AAL5, UDP, and Myricom modules. We also thank John Anderson, Robert Jacob, and Chad Schafer for making the coupled model available to us, and Michael Plastino for performing the coupled model measurements. This work was supported by the National Science Foundation's Center for Research in Parallel Computation, under Contract CCR-8809615, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

# References

[1] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[2] E. Biagioni. A structured TCP in Standard ML. In *Proc. SIGCOMM '94*, 1994. Also as Technical Report CMU-CS-94-171, Carnegie Mellon.

[3] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computing Systems*, 2:39–59, 1984.

[4] R. Braden, L. Zhang, D. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 functional specification. Technical report, ISI/PARC/UCS, 1995. Work in progress.

[5] R. Butler and E. Lusk. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[6] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[7] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*. The MIT Press, 1993.

[8] D. E. Comer. *Internetworking with TCP/IP*. Prentice Hall, 3rd edition, 1995.

[9] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273. ACM, 1993.

[10] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide area visual supercomputing. *International Journal of Supercomputer Applications*, 10(2), 1996.

[11] D. Diachin, L. Freitag, D. Heath, J. Herzog, W. Michels, and P. Plassmann. Remote engineering tools for the design of pollution control systems for commercial boilers. *International Journal of Supercomputer Applications*, 10(2), 1996.

[12] T. L. Disz, M. E. Papka, M. Pellegrino, and R. Stevens. Sharing visualization experiences among remote virtual environments. In *International Workshop on High Performance Computing for Computer Graphics and Visualization*, pages 217–237. Springer-Verlag, 1995.

[13] J. Drake, I. Foster, J. Michalakes, B. Toonen, and P. Worley. Design and performance of a scalable parallel Community Climate Model. *Parallel Computing*, 21(10):1571–1591, 1995.

[14] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[15] I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.

[16] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 1996. To appear.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[18] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Preprint MCS-P567-0296, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

[19] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1995.

[20] C. Lee, C. Kesselman, and S. Schwab. Near-real-time satellite image processing: Metacomputing in CC++. *Computer Graphics and Applications*, 1996. to appear.

[21] C. Mechoso et al. Distribution of a Coupled-ocean General Circulation Model across high-speed networks. In *Proceedings of the 4th International Symposium on Computational Fluid Dynamics*, 1991.

[22] M. Norman et al. Galaxies collide on the I-WAY: An example of heterogeneous wide-area collaborative supercomputing. *International Journal of Supercomputer Applications*, 10(2), 1996.

[23] D. Notkin, A. Black, E. Lazowska, H. Levy, J. Sanislo, and J. Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):259–273, 1988.

[24] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computing Systems*, 10(2):110–143, 1992.

[25] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995. `http://www.supercomp.org/sc95/proceedings/`.

[26] C. Partridge. *Gigabit Networking*. Addison-Wesley, 1994.

[27] B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1993.

[28] C. Tschudin. Flexible protocol stacks. In *Proc. ACM SIGCOMM '91*. ACM, 1991.

[29] T. v. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[30] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proc. Principles of Distributed Computing Conf.*, 1995. http://www.cs.cornell.edu/Info/People/rvr/papers/podc/podc.html.