Managing Multiple Communication Methods in High-Performance Networked Computing Systems*

Ian Foster[†], Jonathan Geisler[*], Carl Kesselman[‡], and Steven Tuecke[*]
`http://www.mcs.anl.gov/nexus/`

Abstract

Modern networked computing environments and applications often require—or can benefit from—the use of multiple communication substrates, transport mechanisms, and protocols, chosen according to where communication is directed, what is communicated, or when communication is performed. We propose techniques that allow 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 explain how communication link and remote service request mechanisms facilitate the specification and implementation of multimethod communication. These mechanisms have been implemented in the Nexus multithreaded runtime system, and we use this system to illustrate solutions to various problems that arise when implementing multimethod communication. We also illustrate the application of our techniques by describing a multimethod, multithreaded implementation of the Message Passing Interface (MPI) standard, constructed by integrating Nexus with the Argonne MPICH library. Finally, we present the results of experimental studies that reveal performance characteristics of multimethod communication, the Nexus-based MPI implementation, and a large scientific application running in a heterogeneous networked environment.

1 Introduction

Increasingly, high-performance applications need to exploit heterogeneous collections of computing resources interconnected via high speed networks. Examples of such applications include coupled modules [30, 31], collaborative environments [9, 10] and computations that couple specialized data sources to supercomputers for processing and visualization [28]. These applications are heterogeneous not only in in their computational requirements, but also in the types of data that they communicate. One significant consequence of this changing environment is an increase in the number of communication methods that can usefully be employed in networked computations. For example, different communications may use different network interfaces, low-level protocols, and data encodings, and may have different quality of service requirements. These developments introduce challenging problems for developers of parallel programming tools. In

*To appear in *J. Parallel and Distributed Computing.*
[†]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
[‡]Beckman Institute, California Institute of Technology, Pasadena, CA 91125.

particular, techniques are required that allow communication operations to be specified independently of the methods used to implement them, and that support both automatic and programmer-assisted method selection. In addition, mechanisms are required for identifying applicable methods and for incorporating multiple communication methods into an implementation.

No existing system addresses all of these issues. Various tools support computing in heterogeneous environments [1, 4, 32], but most do not exploit or expose the heterogeneous nature of the network or applications. The p4 [5] and PVM [20] communication libraries can use different low-level methods in heterogeneous environments, but the set of methods used is not extensible. The x-kernel [33] and Horus [40] allow new protocols to be constructed by composing primitive elements, but do not support automatic discovery or dynamic reconfiguration of communication methods.

In this paper, we introduce new techniques for constructing heterogeneous computations in which communication patterns and the methods used to perform these communications can change dynamically, in both time and space. We first examine the principal factors motivating multimethod communication and present requirements that tools supporting multimethod communication should satisfy. We then explain how a structure called a communication startpoint can provide a concise, mobile representation of both the target of a communication operation and the methods used to perform that operation. We describe a simple algorithm for selecting automatically from among applicable methods when the startpoint is received from a remote location. We also explain how our architecture supports manual selection of communication methods. We describe a single-sided communication mechanism called a remote service request, and explain how this mechanisms can be used to implement both point-to-point and streaming protocols. These techniques have all been implemented in the context of the Nexus multithreaded runtime system [19], and we use the architecture of the Nexus implementation to illustrate the presentation.

In evaluating these techniques, we must necessarily be concerned with both their generality and the efficiency with which they can be implemented. With respect to generality, we note that Nexus has been used to implement a variety of parallel languages and communication libraries [6, 14, 10]. We describe here an implementation of the standard Message Passing Interface (MPI) [24], constructed by adapting the MPICH [23] implementation of MPI to use Nexus communication primitives [16]. This implementation allows an MPI program to execute unchanged in heterogeneous environments, with communication method selected according to default rules, depending on the source and destination of the message being sent. We also explain how the Nexus implementation of MPI benefits from access to Nexus multithreading mechanisms. This MPI implementation was used extensively in the I-WAY wide area computing experiment [8], where it and other Nexus-based tools [6, 10] supported multiple applications on a wide range of networks and computers, including IBM SP2, Intel Paragon, Cray C90, and SGI Power Challenge. We address the question of efficiency by reporting the results of experimental studies using both simple benchmark programs and a large scientific application; these allow us to quantify the performance characteristics of the Nexus implementations of multimethod communication and MPI.

2 Multimethod Communication

We first discuss why multimethod communication is important, develop a set of requirements for an implementation, and describe the communication link and remote service request mechanisms that we use to support multimethod communication.

2.1 Motivation

We review situations in which we may want to support multiple communication methods in a single application. 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 *when* communication is performed.

- *Transport mechanisms.* While the Internet Protocol provides a standard transport mechanism [7], parallel computers and local area networks often support alternative, more efficient mechanisms: for example, shared memory, a vendor-specific communication library such as IBM's Message Passing Library (MPL), or MessageWay over a local Asynchronous Transfer Mode (ATM) switch. In a wide area environment, optimized protocols can be employed in an ordered network. The ability to use a mixture of specialized transport mechanisms and TCP can be crucial to application performance.

- *Network protocols.* Particularly in a wide area environment, we may want to use specialized protocols such as UDP, IP multicast, reliable multicast, and Realtime Transport Protocol (RTP) for selected data, such as shared state updates and video, while at the same time using reliable point-to-point protocols (e.g., TCP/IP) for other data.

- *Quality of service (QoS).* Future ATM-based networks will support channel-based QoS reservation and negotiation [35]. High-performance multimedia applications will likely want to reserve several channels providing different QoS. For example, a multimedia application might use a high-reliability, low-bandwidth channel for control information, and a lower-reliability, high-bandwidth channel for image data.

- *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 [36], 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.

- *Time-varying properties.* Many of the choices listed above can vary over time in both predictable and unpredictable fashions. Users may want to write programs that can adapt to anticipated or unanticipated network outages, or that can take advantage of lower network loads at night or the availability of dedicated networks via reservation systems.

- *Application-specific protocols.* Finally, we note that certain applications may wish to employ specialized protocols for certain data. For example, an application might compress image data when transferring it across the country.

2.2 Requirements

Given the recognition 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. From the user's point of view, the following requirements are of particular importance.

- *Separate specification of communication interface and communication method.* Ease-of-use and portability concerns demand that programmers be able to specify communications using a single abstraction (whether message passing, remote procedure call, etc.), independently of the actual method used to effect a particular communication.

- *Automatic selection.* Ease-of-use and portability concerns also demand that automatic selection mechanisms be provided, so that reasonable performance can be achieved when programmers lack the expertise, motivation, or time to guide communication method selection. Ideally, the rules or heuristics used to guide selection should be easily modified by systems developers or interested programmers.

- *Manual selection.* Developers of performance-critical applications will sometimes require manual selection mechanisms that allow them to obtain information on available methods and override automatic selections. Automatic and manual selection methods need 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.

- *Parameterized methods.* For some communication methods, it will be important to allow programmers to manage low-level behavior by specifying values for key parameters. For example, a TCP protocol might allow a programmer to specify socket buffer sizes.

- *Feedback.* Programmers require method-specific feedback mechanisms if they are to evaluate the effectiveness of automatic selection or tune manual selections. For example, the Realtime Transport Control Protocol used in conjunction with RTP provides feedback on frame loss rate and jitter. Programmers require access this information.

Two additional requirements arise at the implementation level.

- *Environment enquiry.* Implementations of multimethod communication require enquiry functions that they can use to obtain the environmental information needed to determine which methods are applicable in particular situations. For example, shared-memory communication is appropriate only if directed to another process within the same shared address space.

- *Compositionality.* Implementations of multimethod communication must permit the coexistence of multiple methods within a single application. This is a nontrivial problem, as different methods may use quite different mechanisms for initiating and processing communications. Mechanisms are also required for configuring an executable with a particular set of communication methods and/or dynamically loading method implementations.

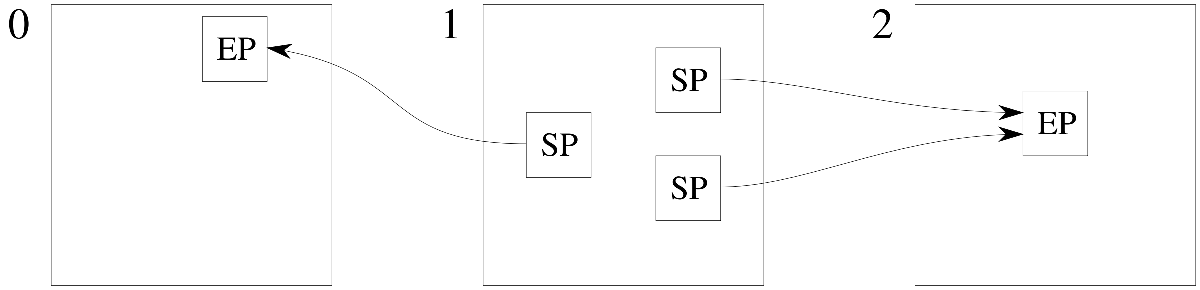

Figure 1: The communication link and its role in communication. The figure shows three address spaces; three startpoints in address space 1 reference endpoint in address spaces 0 and 2.

2.3 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 present an approach based on the communication link and remote service request mechanisms, and explain why we believe this approach to be superior to the most obvious alternative, namely point-to-point communication and communicators.

With our primitives, communication flows from a communication *startpoint* to a communication *endpoint*. (Earlier papers on the Nexus communication primitives [19] used the term *global pointer* to describe a startpoint and provided an implicitly defined endpoint. We introduce the terms startpoint and endpoint because we find them more descriptive.) A startpoint is bound to an endpoint to form a *communication link*. Many startpoints can be bound to a single endpoint, in which case incoming communication is merged as in typical point-to-point message passing systems. Similarly, many endpoints can be bound to a single startpoint, resulting in a multicast communication pattern. Both startpoints and endpoints can be created dynamically; the startpoint has the additional property that it can be moved between processors using the communication operations we now describe.

Communication links are used in conjunction with asynchronous *remote service requests* (RSRs) which invoke actions on remote objects. An RSR is specified by providing a startpoint, an RSR handler identifier and a data buffer, which is constructed using PVM [20] style `put` routines. Issuing an RSR causes the data buffer to be transfered from the startpoint to the bound endpoint, after which the routine specified by the handler is executed, potentially in a new thread of control. Both the data buffer and endpoint-specific data are available to the RSR handler.

Key to the communication link's utility is the mobility of the startpoint. A process can bind a startpoint to a local endpoint and then communicate that startpoint to other processes, providing the other processes with a handle that they can use to perform RSRs back to the local endpoint. A process can create multiple handles, referring to different endpoints, hence allowing communications intended for different purposes to be distinguished.

The communication link mechanism is naturally extended to encapsulate *how* as well as *where* communication is to be performed. It suffices to associate with a startpoint information about the methods that can be used to communicate to the bound endpoint. A process receiving such a startpoint then has all the information required to communicate with the referenced object, even in a heterogeneous system.

Associating communication methods with communication links provides fine-grained control over how communication is achieved. We illustrate this point with three scenarios.

- *Networks with asymmetric bandwidth.* In a cable modem, different methods may be used for incoming and outgoing communication. This situation can be represented explicitly by a pair of communication links.

- *Multi-protocol networks.* Consider a system in which two low-level protocols are available, with one protocol better suited for small, latency-sensitive communications, and the other for large communications. We can represent this situation by creating two communication links to the same address space, with each link defined to use a different protocol. We can then optimize application performance by using one link for synchronization functions and the other for data transfer.

- *Streaming protocols.* Multi-media applications often require specialized, stream-oriented protocols (e.g., MPEG compression) for audio and video data. This requirement can be satisfied by defining a link that uses a stream-oriented protocol. Each RSR on this communication link transfers a block of video data, which is incorporated incrementally and asynchronously into the appropriate buffer in the destination process.

An alternative approach to multimethod communication is to use two-sided message passing primitives, rather than single-sided remote service requests, and to associate method choices with group constructs such as MPI communicators [24]. This is not an unreasonable approach, but is less flexible than communication links and RSRs. We use the three scenarios above to explain why. First, we note that two-sided communication as found in first-generation libraries such as PVM provides, in effect, just a single endpoint per node. This structure complicates solutions to the first and second scenario described above. The second-generation MPI system introduces a communicator mechanism that allows for the creation of unique communication contexts. Each communicator behaves like a separate endpoint in our primitives and can—as we describe below—utilize a specific communication method. However, a communicator must be created by a collective operation and cannot be transferred between nodes, limiting its utility for multimethod communication. In [18], we propose an extension to MPI communicators that overcomes this limitation. However, a second limitation of two-sided communication that cannot be overcome is that the protocol for synchronizing and extracting data at the receive side of the transfer is defined by the communication model. That is, data *must* be extracted by a matching receive. This rigidity hinders implementation of the third scenario.

3 Architecture

We now turn our attention to the techniques used to implement multimethod communication. We describe these techniques in the context of Nexus [19], a portable, multithreaded communication library designed for use by parallel language compilers and higher-level communication libraries. In addition to communication links and remote service requests, Nexus provides support for lightweight threading, which, as we shall explain, can simplify the implementation of multimethod communication. For pragmatic reasons, Nexus thread support is based on a

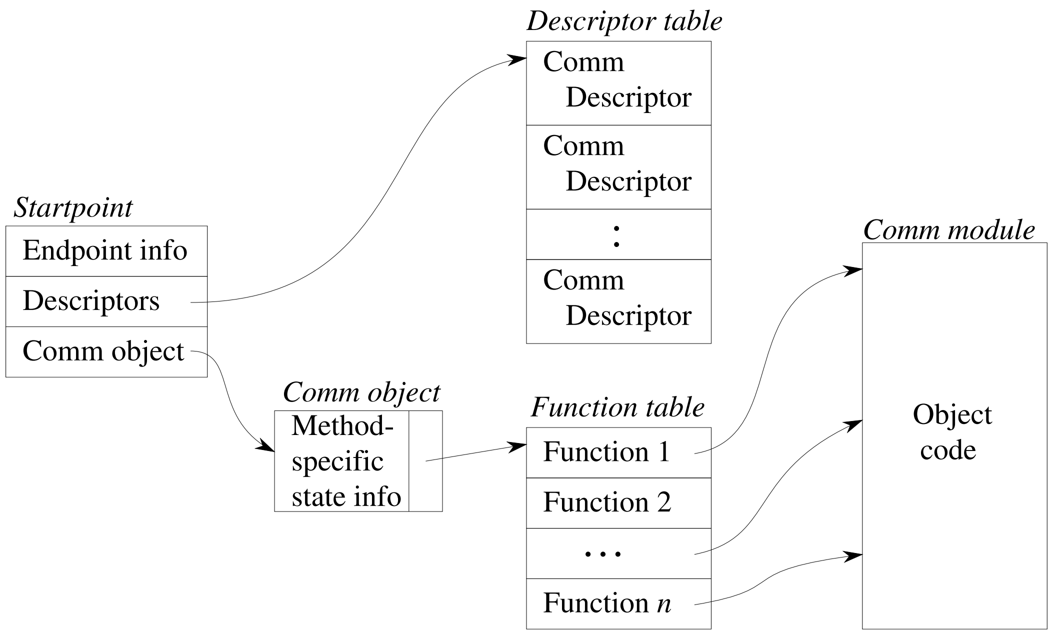

Figure 2: Nexus data structures used to support multimethod communication. The figure shows the startpoint, communication descriptor table, communication object, function table, and communication module. The data structures are explained in the text.

subset of the POSIX threads standard [27]; it supports thread creation, termination, and synchronization. Nexus also supports the ability to create multiple address spaces, or *contexts*, within a single node. On some systems, contexts may be implemented as processes; on others, as separate code and data segments within a single process.

3.1 Multimethod Communication Architecture

We first describe the software architecture used to support multiple communication methods. In particular, we explain how communication methods are represented, how the set of methods available in a particular process is determined, and how communication method information is associated with communication startpoints. In the process, we describe the structures illustrated in Figure 2: the communication module, communication descriptor, communication object, communication function table, and startpoint.

Communication module and function table. A communication method is implemented within Nexus by a *communication module*. A communication module is accessed via a standard interface (see Table 1), which includes communication-oriented functions, an initialization function, enquiry functions to return the method name and parameter information, and functions used to construct communication descriptors and communication objects.

Several methods are provided for determining which communication modules are required in a particular executable. When building the Nexus library, a default set of modules can be defined as a configuration parameter; this information is propagated to the executable as a static table of module names. The composition and ordering of this default set can be overridden by

Table 1: A subset of the Nexus communication module interface

Name	Purpose
`method_name`	Return communication method name
`init`	Initialize communication module
`shutdown`	Shutdown communication module
`increment_reference_count` `decrement_reference_count`	Reference counting on communication object
`communication_descriptor`	Construct communication descriptor
`communication_object`	Construct communication object
`poll`	Nonblocking poll
`blocking_poll`	Blocking poll
`init_remote_service_request`	Initiate remote service request
`set_buffer_size`	Set RSR buffer size
`send_remote_service_request`	Perform RSR

entries in an external resource database. Additional information can be provided via command line arguments or function calls.

The Nexus system incorporates communication modules into a computation by first invoking an enquiry function on each required module. This function returns a *function table* that provides access to the module's implementation of the standard interface functions (Figure 2). The initialization function included in this table is invoked subsequently to perform any initialization required to use the module. For example, a module that uses TCP/IP communication might use the initialization component to set up a `select` call to detect incoming communications.

Communication descriptor. Each communication module defines a function that returns a *communication descriptor* containing the information required to access the context in which the function is called, using the communication method in question. For example, when using MPL to communicate between nodes on IBM SP multicomputers, we require both a node number and a globally unique session identifier. The unique identifier is used to distinguish between different SP partitions, and can be constructed from the hostname, the IP address of node 0 in the partition, and the time on node 0. On the Intel Paragon, the descriptor also includes the name of the process with which we want to communicate. This is because on the Paragon, several parallel computations can execute concurrently on the same processors.

As these examples show, the size of the communication descriptor depends on the method used, but is normally small: three to five words is typical. Table 2 shows the information contained in this descriptor for a variety of communication methods.

Communication descriptor table. A communication descriptor contains all the information required to communicate with a particular context using a particular method. Hence, a context can create a complete specification of the various methods that it supports, simply by creating one communication descriptor for each of its communication modules. A *communication descriptor table* is a concise and easily communicated representation of this information, in which the various descriptors are concatenated as a contiguous byte array.

Table 2: Contents of the communication descriptor for different communication methods

Type	Communication descriptor contents
IBM SP2 (MPL)	Node #, unique session identifier
Intel Paragon (Nx)	Node #, unique session identifier, process id
TCP/IP	Host name, port #
Shared memory	Machine name, shmem pool addr, queue entry in pool

Communication object. Each communication module defines a function that can be applied to a communication descriptor to obtain a *communication object*. This structure represents an active connection involving the associated method. It contains the information contained in the communication descriptor, plus a pointer to the associated function table and any state information needed to represent the active connection. For example, a communication object for a TCP connection would contain the file descriptor for the TCP socket. The communication object also contains a reference to the function table created when the communication module was initialized.

Communication objects are shared among startpoints that reference the same context and use the same communication method.

Startpoint. As shown in Figure 2, a startpoint contains endpoint information, plus pointers to the associated communication descriptor table and communication object. The communication object represents the communication method that is currently in use on this startpoint, while the descriptor table represents the communication methods supported by the remote context.

The information contained in communication descriptor tables is propagated between contexts as follows. To create a startpoint from a context A to an endpoint X in a second context B, we first create a startpoint bound to X within context B, and then use an RSR to communicate the new startpoint to context A. (The bootstrapping problem of obtaining an initial link from B to A is addressed by Nexus context creation functions, which return startpoints bound to default endpoints in newly created contexts.)

When a startpoint is created in a context, the communication descriptor table representing the methods supported by that context is attached to the startpoint. When the startpoint is communicated to another context (e.g., from B to A), this descriptor table is passed with it. Hence, any context receiving a startpoint also receives the information required to communicate to the referenced endpoint. The context must then select one of the methods specified in the descriptor table, and use this 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.

The startpoint implementation that we have described is very general, but makes startpoints rather heavyweight entities. This is acceptable in a wide area context, where the cost of communicating a few tens of bytes of descriptor table is insignificant. However, it can be unacceptable in more tightly coupled systems. The Nexus implementation uses two different optimizations to minimize replication and communication of descriptor data:

- *Homogeneous communication.* Contexts created as part of a fixed-sized partition within a single parallel computer will typically use the same communication methods. In these situations, a single descriptor table can be assembled and broadcast to all contexts when the partition is created. No descriptor table need be attached to individual startpoints, reducing storage requirements and communication costs. If the startpoint is passed to an external context, an appropriate descriptor table must be attached.

- *Heterogeneous communication with default descriptor table.* Each context in a Nexus computation defines a default descriptor table. The first time a context A passes a startpoint involving this default table to another context B, it attaches the default table; on subsequent occasions, A omits the table, signaling that B should use the previously communicated default.

These optimizations ensure that a distinct descriptor table need be attached to a startpoint only when an application creates a nonstandard descriptor table, whether by adding specialized method selection methods, reordering the methods on the table, or adding or deleting methods.

3.2 Selecting a Communication Method

We explained earlier how a startpoint received from another context has associated with it a descriptor table identifying the methods supported by the referenced endpoint. Upon receipt of a startpoint, a context must determine which of these methods is to be used for subsequent communication using that startpoint. As explained in Section 2.2, we want to support both automatic and manual method selection.

The current Nexus implementation uses a simple automatic selection rule: it selects the first applicable method detected in a linear scan of a received descriptor table. A method is "applicable" if it is supported by the context that receives the startpoint. The function call applied to a descriptor to create a communication object also checks for applicability, and returns a null value if the method is not applicable. The nature of the check is method dependent. For example, if both the checking context and referenced context support MPL, then we must also check that the two contexts are in the same partition on the same IBM SP. The method descriptor contains the information required to perform this test.

Because we select communication methods by means of a linear scan of the communication descriptor table, it suffices to order methods from "fastest" to "slowest" to ensure that the "fastest" applicable method is always selected. The user can then influence the choice of method by the communication descriptor table, for example by reordering entries or by adding or deleting descriptors. Adding a descriptor causes the associated communication module to be loaded if it is not already resident, while deleting a descriptor prevents subsequent selection of the associated communication module.

We note that the selection method just described is suboptimal in environments in which different methods are "faster" in different situations. For example, as noted above a "slower" method might actually be faster for a particular message size. Or, while one network may be "fastest" in terms of raw bandwidth, instantaneous network load may make some "slower" network a better choice at a particular time. The framework that we describe can easily be extended to support more sophisticated rules; the definition of such rules is a subject of current research.

Figure 3 illustrates the techniques used to determine communication method selection. The figure illustrates 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 startpoint was received from node 2, its attached

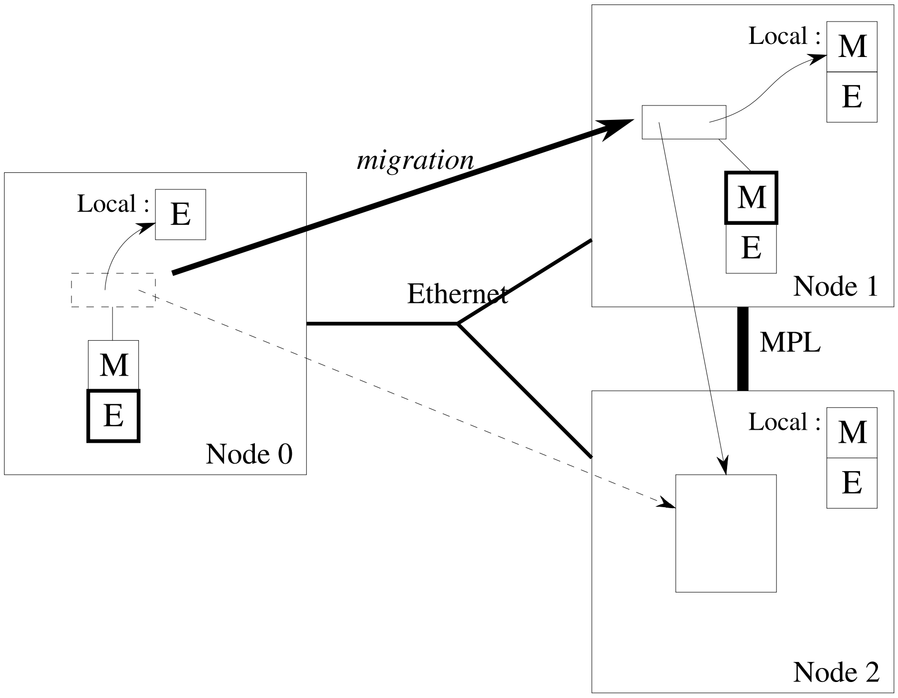

Figure 3: Communication method selection in Nexus. See text for details.

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 supported by both nodes and that both nodes are on the same SP partition. Hence, the faster MPL is used.

3.3 Security as a Communication Method

Much of the discussion above has focused on the selection process as a choice between alternative communication substrates. However, the advantages of multimethod communication extend beyond communication protocols. We can use the same techniques to control other aspects of communication, such as security.

In [17], we show how our multimethod communication architecture can be used to support applications in which certain communication operations must encrypt data before sending it over an open network. For these applications, it suffices to create an encrypting communication method. (To avoid an explosion in the number of communication methods, Nexus allows an arbitrary data transformation to be applied as part of a communication method. Encryption can be implemented as one of these transformations.) The encrypting method can be applied selectively, as can other communication methods.

3.4 Processing at Destination

In the preceding discussion, we have described the techniques used to represent and select communication methods when initiating a communication. We now address the question of how remote service requests (RSRs) are processed at the endpoint. We must first explain some details regarding how Nexus handles messages. A Nexus context must typically be prepared to process RSRs received from multiple sources and communicated with different methods. Frequently, different sources or communication methods may require different operating system (OS) mechanisms, although multiple sources and methods can also be multiplexed onto a single mechanism. For example, on an IBM SP2, specialized `mpc_status` and `mpc_recv` calls are used to detect and receive incoming MPL messages, while Unix `select` and `read` calls are used for TCP communications.

The techniques used to detect and process incoming RSRs have to trade off fast RSR response time against overheads incurred at the destination processor [19]. Threads can simplify implementation. If an OS allows a thread to block on a system call, then a specialized communication server thread may be created for each OS mechanism. This thread will be enabled only when an RSR is available. If an OS does not provide this capability, we use explicit probe operations performed by a single communication server thread. Various combinations of round-robin scheduling, priorities, and explicit yields in user code can provide some degree of control over the frequency with which the server thread is scheduled [19].

When using probe operations to detect RSRs, complex tradeoffs can arise if probes for different communication methods have very different costs. For example, on many MPPs, the probe operation used to detect communication from another processor is cheap, but a TCP `select` is expensive. (On the SP2, `mpc_status` and `select` cost around 10 and 100 μsecs, respectively.) Our current Nexus implementation addresses these issues by causing more expensive probes to be performed less frequently than inexpensive probes. We examine the performance implications of this technique in Section 5. We can also imagine an adaptive algorithm that varies polling frequency according to observed RSR frequency from different sources. Or, we can introduce a dedicated proxy process responsible solely for receiving incoming messages on the "slow" mechanism; this proxy can then forward them to other processes using the "fast" mechanism.

Special techniques are required when multiple communication methods are multiplexed over a single OS mechanism. For example, if we define a communication method that compresses communicated data, then a destination process may receive RSRs communicated using both the regular uncompressed method and the specialized compressed method. These cases can be addressed by referring to the endpoint specified in the message; the endpoint structure will indicate whether the data should be uncompressed.

4 An MPI Implementation

In previous sections, we have sought to demonstrate that communication links, remote service requests, and multithreading are convenient mechanisms for specifying multimethod communication. We must now address the question of whether these mechanisms are useful for practical parallel programming tasks. We do so by demonstrating that Nexus mechanisms can be used to construct an implementation of the widely-used Message Passing Interface (MPI) standard. This implementation provides both multimethod communication and multithreading in an MPI context and, as we show in Section 5, has good performance characteristics.

We emphasize that an implementation of MPI is not the only application of Nexus mechanisms; it is certainly not the programming model for which Nexus is best suited. Other systems that use Nexus facilities include parallel object-oriented languages (for example, CC++ [6] and Fortran M [14]), parallel scripting languages (nPerl), and communication libraries (CAVEcomm [10] and a Java library). We consider MPI here because it is a well-known model, and also because it might appear that single-sided communication is ill-suited to implementing MPI's two-sided communication.

4.1 MPI and MPICH

We first review important features of MPI and of the MPICH implementation on which this work is based.

The Message Passing Interface defines a standard set of functions for interprocess communication [24]. It defines functions for sending messages from one process to another (point-to-point communication), for communication operations that involve groups of processes (collective communication, such as reduction), and for obtaining information about the environment in which a program executes (enquiry functions). The communicator construct combines a group of processes and a unique tag space, and can be used to ensure that communications associated with different parts of a program are not confused.

MPICH [23] is a portable, high-performance implementation of MPI. It is structured in terms of an abstract device interface (ADI) that defines low-level communication-related functions that can be implemented in different ways on different machines [21, 22]. The Nexus implementation of MPI is constructed by providing a Nexus implementation of this device. The use of the ADI simplifies implementation, but has some performance implications, which we discuss below.

4.2 Implementing the Abstract Device Interface

Figure 4 illustrates the structure of the MPICH implementation of MPI. Higher-level functions such as those relating to communicators and collective operations are implemented by a device-independent library, defined in terms of point-to-point communication functions provided by the ADI. To achieve high performance, the ADI provides a rich set of communication functions supporting different communication modes. A typical implementation of the ADI will map some functions directly to low-level mechanisms, and implement others via library calls. The mapping of MPICH functions to ADI mechanisms is achieved via macros and preprocessors, not function calls. Hence, the overhead associated with this organization is often small or nonexistent [23].

The ADI provides a fairly high-level abstraction of a communication device: for example, it assumes that the device handles the buffering and queuing of messages. The lower-level channel interface defines simpler functions for moving data from one processor to another. For example, it defines `MPID_SendControl` and `MPID_SendChannel` functions that can be used to implement the MPI function `MPI_Send`. On the destination side, the test `MPID_ControlMsgAvail` and function `MPID_RecvAnyControl` are provided, and can be used to implement `MPI_Recv`. Different protocols can be selected; the best in many circumstances sends both the message envelope (tag, communicator, etc.) and data in a single message, up to a certain data size, and then switches to a two-message protocol so as to avoid copying data.

The Nexus implementation of the channel device implements channel device send functions as RSRs to "enqueue message" handlers; these handlers place data in appropriate queues, or copy directly to a receive buffer if a receive has already been posted. As this brief description shows, the mapping from ADI to Nexus is quite direct; the tricky issues relate mainly to avoiding extra copy operations. The principal overheads relative to MPICH comprise an additional

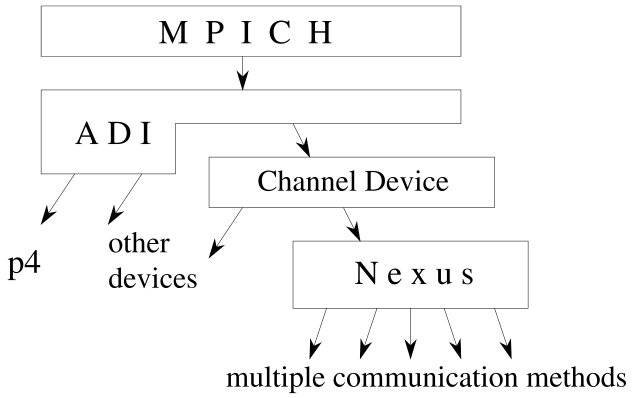

Figure 4: The Nexus implementation of MPI is constructed by defining a Nexus instantiation of the MPICH channel device, a specialization of the abstract device interface.

32 bytes of Nexus header information (we expect to reduce this to 16 bytes in the near future), which must be formatted and communicated; the decoding and dispatch of the Nexus handler on the receiving node; and a small number of additional function calls. We quantify these costs in Section 5. Most are artifacts of the channel device, and will be avoided in the near future by a redesign of the MPICH ADI.

Finally, we observe that the Nexus implementation of MPI is structured so that Nexus thread management functions and MPI communication functions can both be used in the same program. This coexistence is simplified by the fact that the MPI specification is *thread safe*. That is, there is no implicit internal state that prevents the execution of MPI functions from being interleaved. The Nexus library addresses other thread safety issues, ensuring that only one thread at a time accesses nonthread-safe system components, such as communication devices and I/O libraries on many systems.

4.3 MPI Added Value

The Nexus implementation of MPI provides three benefits over and above those provided by MPICH: multimethod communication, interoperability with other Nexus applications, and multithreading.

The automatic selection of communication methods is supported directly in the Nexus implementation of MPI. An interesting question is how to support manual control of method selection in an MPI framework. We propose that this be achieved via MPI's caching mechanism, which allows the programmer to attach to communicators, and subsequently modify and retrieve, arbitrary key/value pairs called *attributes*. An MPI implementation can be extended to recognize certain attribute keys as denoting communication method choices and parameter values. For example, a key `TCP_BUFFER_SIZE` might be used to specify the buffer size to be used on a particular communicator.

A second benefit that accrues from the Nexus implementation of MPI is interoperability with other Nexus-based tools. For example, in the I-WAY networking experiment [8], numerous applications used the CAVEcomm [10] client-server package to transfer data among one or more virtual reality systems and a scientific simulation running on a supercomputer. When the simulation itself was developed using MPI, the need arose to integrate the polling required to detect communication from either source. This integration was achieved within Nexus, as described in Section 3.4.

The third benefit that accrues from the use of Nexus is access to multithreading. The concurrent execution of multiple lightweight threads within a single process is a useful technique for masking variable latencies, exploiting multiprocessors, and providing concurrent access to shared resources. Various approaches to the integration of multithreading into a message-passing framework have been proposed [2, 12, 18, 37, 39, 13, 25, 42]. The Nexus implementation of MPI supports a particularly simple and elegant model that does not require that explicit thread identifiers be exported from MPI processes. Instead, threads are created and manipulated with Nexus functions, and inter-thread communication is performed using standard MPI functions, with tags and/or communicators being used to distinguish messages intended for different threads. The MPI/Nexus combination can be used to implement a variety of interesting communication structures. For example, we can create two communicators and communicate independently on each from separate threads, using either point-to-point or collective operations. Or, several threads can receive on the *same* communicator and tag value. In a multiprocessor, the latter technique allows us to implement parallel servers that process requests from multiple clients concurrently. Nexus support for dynamic resource management and multithreading also provides a framework for implementing new features proposed for MPI-2, such as dynamic process management, single-sided communication, and multicast.

The multithreaded MPI also has its limitations. In particular, it is not possible to define a collective operation that involves more than one thread per process. This functionality requires extensions to the MPI model [18, 26, 37].

5 Performance Studies

We have conducted a variety of experiments to evaluate the performance of both our multimethod communication mechanisms and the Nexus implementation of MPI. All experiments were conducted on the Argonne IBM SP2, which is configured with Power 1 rather than the more common Power 2 processors. These processors are connected via a high-speed multistage crossbar switch and are organized by software into disjoint partitions. 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. Both MPL and TCP operate over the the high-speed switch and can achieve maximum bandwidths of about 36 and 8 MB/sec, respectively. TCP communications incur the high latencies typically observed in other environments, and so multiple SP partitions can be used to provide a controlled testbed for experimentation with multimethod communication in networked systems.

5.1 Multimethod Communication Performance

Our first experiments evaluate the performance of the Nexus implementation of multimethod communication. Our benchmark program, `pphandle`, simply 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 message latency for a variety of message sizes. Message transfer is effected by an RSR to the remote node, with the RSR handler invoking an RSR back on the originating node. This extremely simple code typifies the behavior of a program placing data in a remote location, as no new threads are created to execute handlers. We note that this single-threaded, synchronous scenario represents a worst case situation for a multithreaded, one-sided communication system, in that threads cannot be used to advantage and a native message-passing code has complete knowledge of when data will arrive. To provide a basis for comparison, we also evaluate an MPL program that implements the same communication pattern using point-to-point communication.

We measure performance for `pphandle` using both Nexus and a single-threaded version of Nexus called NexusLite. (The former case corresponds to test case *H-to-H* in [19].) In NexusLite, user programs cannot create multiple threads, and probe operations are performed within the computation thread rather than within a separate communication server thread. In addition, there is no need to protect communication operations with locks to ensure mutual exclusion. Hence, NexusLite results provide insights into multithreading costs.

For Nexus, we measure `pphandle` performance over MPL both in a system in which only MPL communication is supported, and in a system in which both MPL and TCP are supported. In the latter case, the `select` calls used to check for pending TCP communications introduce additional overhead, and so we measure performance for a variety of TCP probe frequencies.

Figure 5 shows the results obtained for the experiments just described. Looking first at small message times, we see that the MPL program is the fastest, taking 61.4 μsec for a zero-length message. NexusLite and Nexus take 82.8 and 112.4 μsec respectively when configured to use MPL communication only. We have documented Nexus overheads elsewhere [19]. Briefly, the principal sources of the 21.4 μsec difference between NexusLite and MPL are the setup and communication of the 32-byte header contained in a Nexus message (about 8 μsec) and the lookup and dispatch of the handler on the receive side (about 7 μsec). The additional 29.6 μsec overhead associated with full Nexus is due to locking needed for thread safety and the use of a probe rather than a blocking receive to detect the incoming message.

The upper two lines in the left-hand graph are for NexusLite and Nexus when configured to support TCP as well as MPL communication. These data show the impact of the slower TCP on MPL performance. For a zero-byte message, costs are 156.1 μsec and 188.8 μsec, respectively: about 75 μsec more than the corresponding times without TCP. Because incoming RSRs may arrive on either a socket or in an MPL message, both Nexus and NexusLite probe for messages from both sources. If the MPL probe fails the first time, the more expensive `select` call must be made before a second MPL probe is performed, which has the effect of increasing the average time that it takes to detect availability of an MPL message.

Looking at the large message sizes, we see that NexusLite overheads become insignificant for larger messages. In the threaded Nexus, overheads remain; we believe that we can eliminate these, but have not yet completed this investigation. Of greater interest is the fact that the inclusion of TCP support continues to degrade MPL communication performance even for large messages. We hypothesize that this is because repeated kernel calls in the user code (due to `select` calls) slow down the process of transferring data from the IBM SP2's communication device to user space.

We conducted a second set of experiments using a benchmark `ppmulti` in which two instances of `pphandle` run concurrently (Figure 6). One instance executes the `pphandle` algorithm over MPL while the other executes it over TCP. The two programs execute until the MPL `pphandle` has performed a fixed number of roundtrips, and the one-way latency for each pair is then computed. The experiment was repeated for a range of TCP polling frequencies, expressed in terms of a parameter `skip_poll`, denoting the number of `select` calls skipped between each poll. Figure 7 shows the results of these experiments. The performance of the MPL instance of `pphandle` is degraded significantly by the concurrently executing TCP instance. As

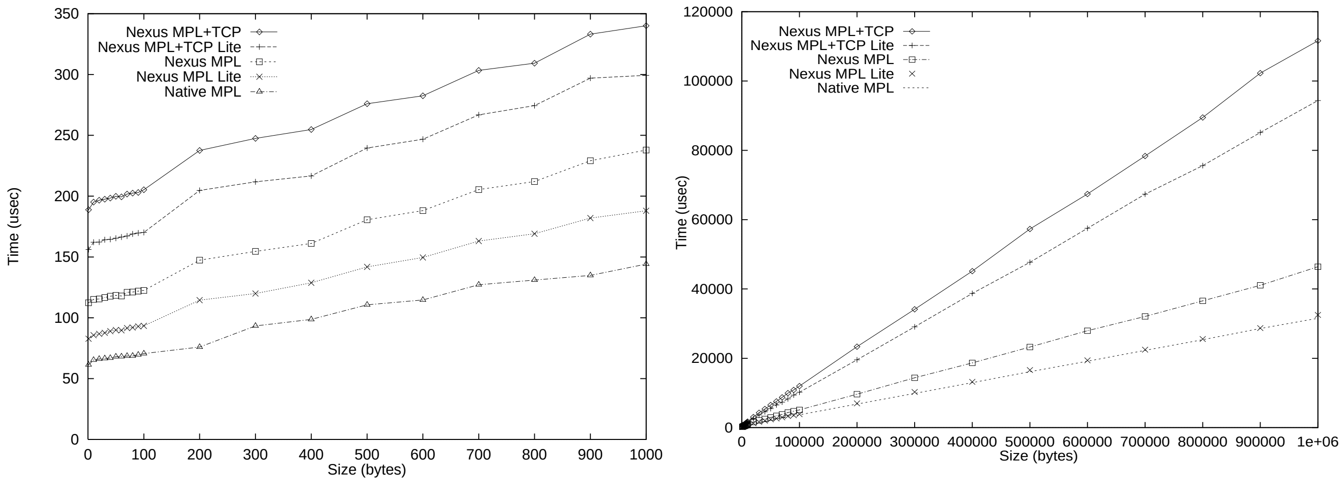

Figure 5: One-way message latency as a function of message size, as measured with both a low-level MPL program and the `pphandle` benchmark, using various versions of Nexus. On the left, we show data for message sizes in the range 0–1000, and on the right a wider range of sizes. See the text for details.

we might expect, MPL performance improves with increasing `skip_poll`, while TCP performance degrades. Interestingly, a modest `skip_poll` value provides a significant improvement in MPL performance, while not impacting TCP performance very badly.

5.2 MPI Performance

We next report on experiments that evaluate the performance of the Nexus implementation of MPI. We used the MPI `mpptest` program [23], which incorporates a "ping-pong" benchmark equivalent to `pphandle`. We executed this program using MPICH and with the Nexus implementation of MPI, in the latter case evaluating NexusLite and Nexus, both with MPL support only, and with MPL and TCP support. Figure 8 shows our results.

The graph on the left shows that MPICH takes 83.8 μsec for a zero-length message. This is comparable with the 82.8 μsec achieved by `pphandle`, suggesting that MPICH and NexusLite are implemented at a similar level of optimization. The NexusLite implementation of MPI incurs an overhead of around 60 μsec for a zero-length message; the graph on the right shows that for larger messages, the overhead becomes insignificant. We have outlined the sources of these overheads in Section 4.2; as we note there, most can be eliminated by improving the MPICH ADI. The jump in the MPICH numbers at 200 bytes is an artifact of the protocols used in the low-level MPL implementation, and is visible in this graph but not in Figure 5 because we plot more points here. Notice the corresponding jump in the Nexus plots at around 170 bytes; the offset is due to the additional header information associated with a Nexus RSR.

5.3 Climate Model Performance

In our final experiment, we used a substantial, communication-intensive parallel application to provide a real-world evaluation of both MPICH/Nexus and our multimethod communication

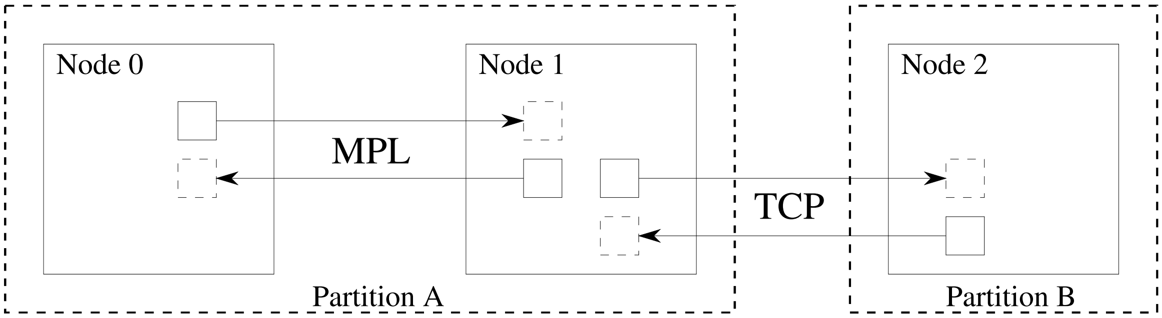

Figure 6: Configuration for the `ppmulti` communication benchmark. Two `pphandle` programs run concurrently, one within an IBM SP2 partition using MPL, and the other between two partitions, using TCP.

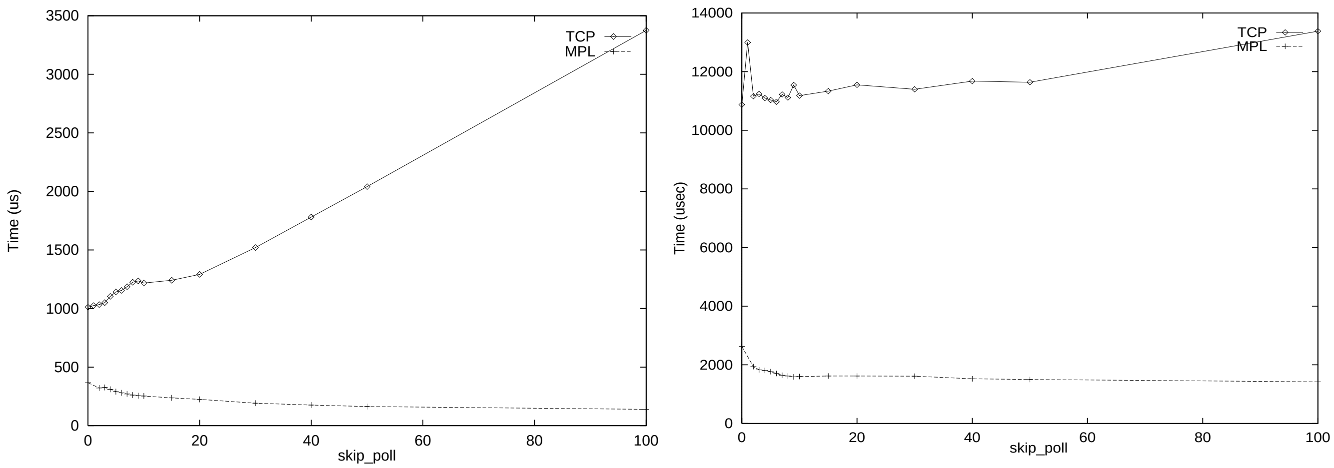

Figure 7: One-way message latency as a function of `skip_poll` for the MPL and TCP instances of `pphandle` executed in the `ppmulti` benchmark described in the text. The graph on the left is for zero-length messages, and the graph on the right is for 10 KB messages.

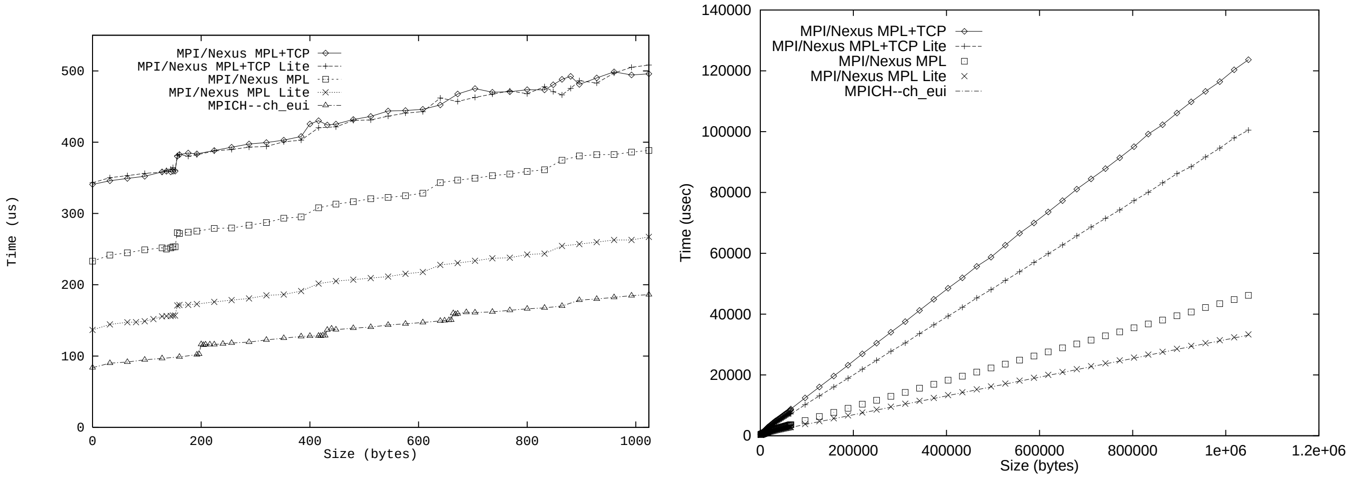

Figure 8: One-way message latency as a function of message size, for various implementations of MPI described in the text. The two graphs show results for small and large messages, respectively.

support. The application is a climate model that couples a large atmosphere model (the Parallel Community Climate Model [11]) with an ocean model (from U. Wisconsin). In brief, the two components execute concurrently, perform considerable internal communication, and periodically exchange boundary information such as sea surface temperature and various fluxes. In the configuration used for these experiments, the atmosphere model operates with its time step accelerated by a factor of six, and runs on 16 processors while the ocean model runs on 8 processors. Information is exchanged every two atmosphere time steps (every one ocean time step).

We measured execution times for the coupled model when using MPICH over MPL, MPI on NexusLite with MPL support only, MPI on NexusLite with TCP support only, and MPI on NexusLite with both MPL and TCP support. For the latter case, we measured performance both in one partition and in two partitions, with the two-partition run placing the two model components in separate partitions (Figure 9) so that MPL was used within each component and TCP for intercomponent communication. The two-partition configuration is a good approximation to a heterogeneous system comprising two IBM SP2's connected by a fast network. Results are provided in the MPI on NexusLite case for a variety of different polling management strategies. In the "manual" strategy, polling is explicitly disabled except when in the coupler; hence, this represents a best case. In the "forwarder" case, a proxy processor (Section 3.4) is used to handle TCP communications (see [15] for details). Finally, we present results for a variety of `skip_poll` values.

Results are presented in Table 3. We see that considerable benefits result from the use of multimethod communication: time per day is 51.5 secs (in an "optimal" case) vs. 65.9 secs when only TCP is used. We also see that an appropriate choice of polling interval allows us to come close to the "optimal" value (a value of 1000 seems to work well in this application), while the use of a proxy is better than a naive polling approach, but less efficient than a selective polling scheme.

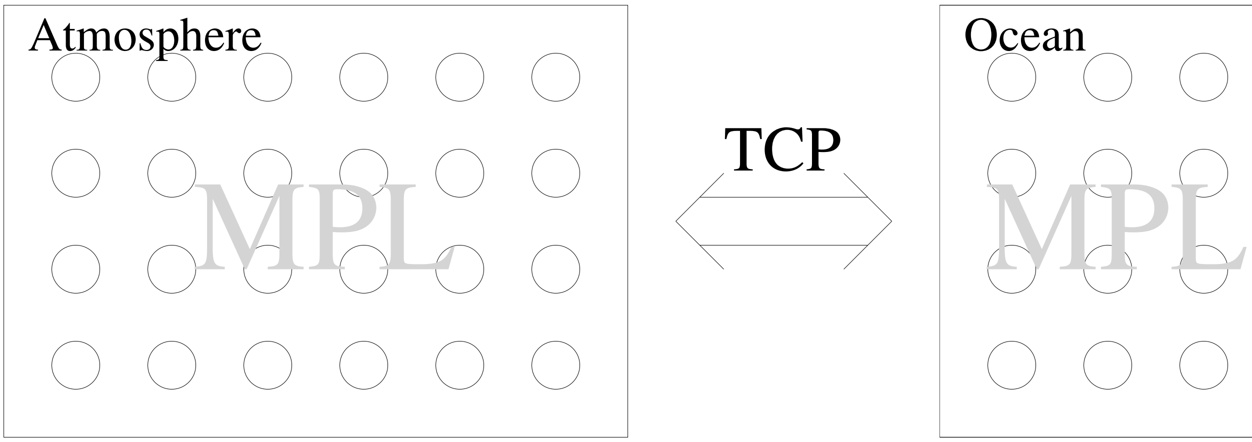

Figure 9: The Argonne/Wisconsin coupled ocean/atmosphere model in the configuration used for our multimethod communication experiments, showing the two IBM SP partitions.

Table 3: Total execution time for the coupled model in different scenarios. Times are in seconds per simulated day on 24 processors. See text for details

Scenario	polling management	Configuration 1 partition	2 partitions
MPICH/MPL	–	49.3	–
Nexus/TCP	–	65.9	–
Nexus/MPL+TCP	manual	–	51.5
Nexus/MPL+TCP	forwarder	–	53.9
Nexus/MPL+TCP	0	58.0	60.0
Nexus/MPL+TCP	1	55.5	55.9
Nexus/MPL+TCP	2	54.4	54.8
Nexus/MPL+TCP	5	53.7	53.7
Nexus/MPL+TCP	20	55.1	53.2
Nexus/MPL+TCP	100	52.5	53.3
Nexus/MPL+TCP	1000	52.3	52.4
Nexus/MPL+TCP	7000	52.8	51.8
Nexus/MPL+TCP	10000	52.8	52.9

6 Related Work

Many researchers have proposed and investigated communication mechanisms for heterogeneous computing systems (for example, [1, 4, 32]). 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, the Intel Paragon implementations of p4 and PVM support heterogeneous computing by using the NX communication library for internal communication and TCP for external communication [5, 20]; p4 supports NX and TCP within a single process, while PVM uses a proxy process for TCP. In both systems, the choice of method is hard coded and cannot be extended or changed without substantial reengineering.

The x-kernel [33] and the Horus distributed systems toolkit [40] both support the concurrent use of different communication methods. In Horus, the primary motivation for multimethod communication is to support various group communication mechanisms in a way that allows applications to pay only for the services that they use [41]. The choice of method is associated with a group, and typical methods include reliable and unreliable multicast, and ordered and unordered delivery. 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 [38] and the Fox project [3] have explored similar concepts and report similar results.

Active Messages (AM) [29] and Fast Messages (FM) [34] are communication systems based on asynchronous handler invocation mechanisms similar to those used in Nexus. The latest AM specification introduces an endpoint construct with some similarities to the Nexus endpoint. However, the AM endpoint is a more heavyweight structure, incorporating both startpoint and endpoint functionality. Another significant difference between AM and Nexus is that AM handlers are used in request/reply pairs, rather than in a one-sided fashion as in Nexus. FM, like Nexus, does not couple sender and receiver. However, it does not have any concept of endpoint. Neither AM nor FM supports heterogeneity; both assume that a computation takes place over a homogeneous, switched network. Thus Nexus is significantly different in scope from both AM and FM.

7 Conclusions

We have described techniques for representing and implementing multimethod communication in heterogeneous environments. These techniques 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. A remote service request mechanism allows point-to-point communication, remote memory access, and streaming protocols to be supported within this framework.

We have used the example of the Nexus runtime system to illustrate the implementation of the various techniques described in this paper. We also discuss how Nexus mechanisms can be used to produce a multimethod, multithreaded implementation of the standard Message Passing Interface (MPI). Performance studies with both Nexus and the Nexus implementation of MPI provide insights into the costs associated with multimethod communication mechanisms.

The results reported in this paper suggest several directions for future work. An immediate priority is to gain practical experience with additional communication methods. Communication modules currently supported include local (intra-context), TCP socket, Intel NX message-passing, IBM MPL, AAL-5 (ATM Adaptation Layer 5), Myricom, unreliable UDP, shared memory, and encryption mechanisms. Streaming protocols and multicast will be considered next. While preliminary design work suggests that they fit the model well, practical experience may suggest refinements.

We also note that Nexus performance can be refined further. The results presented here are promising, in that they show that overheads associated with multimethod communication are small and manageable. However, we know that these overheads can be reduced still further. In particular, the only unavoidable overheads associated with the Nexus implementation of MPI seem to be the few microseconds associated with handler dispatch and the use of probe rather than blocking receive.

A third area of future investigation relates to the techniques used to select communication methods. We 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 developed AAL5, UDP, and Myricom modules. Our work on MPI implementation was made possible by the outstanding MPICH implementation constructed by Bill Gropp, Ewing Lusk, Nathan Doss, and Tony Skjellum; we are grateful for their considerable help with this project. 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] R. Bhoedjang, T. Rumlhl, R. Hofman, K. Langendoen, and H. Bal. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, September 1993.

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] E. Felton and D. McNamee. Improving the performance of message-passing applications by multithreading. In *Proc. 1992 Scalable High Performance Computing Conf.*, pages 84–89. IEEE Computer Society Press, 1992.

[13] A. Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. Technical Report CSTR-940802, University of Virginia, 1994.

[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, J. Geisler, C. Kesselman, and S. Tuecke. Multimethod communication for high-performance metacomputing applications. In *Proceedings of Supercomputing '96*. ACM, 1996.

[16] I. Foster, J. Geisler, and S. Tuecke. MPI on the I-WAY: A wide-area, multimethod implementation of the Message Passing Interface. In *Proceedings of the 1996 MPI Developers Conference*, pages 10–17. IEEE Computer Society Press, 1996.

[17] I. Foster, N.T. Karonis, C. Kesselman, G. Koenig, and S. Tuecke. A secure communications infrastructure for high-performance distributed computing. Preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

[18] I. Foster, C. Kesselman, and M. Snir. Generalized communicators in the Message Passing Interface. In *Proceedings of the 1996 MPI Developers Conference*, pages 42–49. IEEE Computer Society Press, 1996.

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

[20] 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.

[21] W. Gropp and E. Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Preprint MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1994.

[22] W. Gropp and E. Lusk. MPICH working note: Creating a new MPICH device using the channel interface. Technical Report ANL/MCS-TM-213, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1995.

[23] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Technical Report ANL/MCS-TM-213, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

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

[25] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing '94*, pages 350–359, 1993.

[26] M. Haines, P. Mehrotra, and D. Cronk. Ropes: Support for collective operations among distributed threads. Technical Report 95-36, Institute for Computer Application in Science and Engineering, 1995.

[27] IEEE. IEEE P1003.1c/D10: Draft standard for information technology – portable operating systems interface (POSIX), September 1994.

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

[29] A. Mainwaring. Active Message applications programming interface and communication subsystem organization. Technical report, Dept. of Computer Science, UC Berkeley, Berkeley, CA, 1996.

[30] 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.

[31] 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.

[32] 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.

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

[34] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (fm) for Myrinet. In *Proceedings of Supercomputing '95.* IEEE Computer Society Press, 1996.

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

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

[37] A. Skjellum, N. Doss, K. Viswanathan, A. Chowdappa, and P. Bangalore. Extending the message passing interface. In *Proc. 1994 Scalable Parallel Libraries Conf.* IEEE Computer Society Press, 1994.

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

[39] 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.

[40] 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.

[41] R. van Renesse, T. Hickey, and K. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR94-1442, Cornell University, 1994.

[42] D. A. Wallach, W. C. Hsieh, K. Johnson, M. F. Kaashoek, and W. E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. Technical report, MIT Laboratory for Computer Science, 1995.