A Symmetrical Communication Interface for Distributed-Memory Computers

Peter A. Steenkiste

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

We describe a communication interface that presents applications with a simple model of the interlying distributed-memory system. All communication operations have the same regular format, and the send and receive operations are completely symmetrical. For example, both sends and receives can operate on both local and remote buffers. Although this communication model does not correspond directly to the low-level communication primitives supported by the hardware, it can be implemented efficiently, and it gives the users more control over how and when transfers over the network takes place. The interface is the lowest-level communication interface for the Nectar multicomputer.

1. Introduction

Nectar is a high-performance multicomputer [2]. The Nectar communication interface, or Nectarine, was designed to support both applications written by programmers and generated by tools efficiently. The interface should give applications efficient access to the network and should allow them to control how data is transferred. The latter is especially important for regular applications where communication can be scheduled.

In Nectarine, the features of both send and receive operations (address, message description and behavior) are defined by the same set of orthogonal parameters. As a result, the communication primitives are very regular, and the send and receive operations are completely symmetrical. A library implementation of the Nectar interface, with C as the host language, has been in use for two years. Since the design of the interface was driven by application requirements, it is an attractive interface for multicomputers in general.

This paper first describes and justifies the symmetrical communication interface (Section 2). We then discuss its implementation on Nectar (Section 3).

This research was supported by Defense Advanced Research Projects Agency (DOD) monitored by DARPA/CMO under Contract MDA972-90-C-0035.

2. The communication model

We describe the features of the Nectar communication primitives: symmetrical addressing, the description of the message, and the synchronization behavior.

2.1. Addressing

Applications have very diverse communication requirements. Although individual algorithms often use regular communication patterns, there is little regularity across applications or even across different phases of the same application. For this reason, a low-level communication interface should support the unrestricted, reliable exchange of variable-length messages.

The programming model presented to Nectar programmers consists of **tasks** and **buffers**. Tasks are basically processes and buffers are FIFO data structures that can store a limited number of messages, determined at buffer creation time. Tasks communicate by placing messages in (‘send’) and retrieving messages from (‘receive’) buffers. Buffers are created by the application, so applications can, for example, use different buffers for different types of data.

What makes the above communication model unusual is that tasks can send to and receive from any buffer in the system. Interfaces typically implement receives as local operations, while sends transmit the data over the interconnect. We argue that the flexibility of the symmetrical model is useful. The asymmetry of the traditional communication model reflects the addressing restrictions of the hardware: the destination of a message is always determined by the sender. For applications this is an arbitrary restriction, since the sender sometimes does not know the identity of the task that will eventually consume the message.

Figure 1 shows an example. A master task M distributes work to slave tasks S using buffers T, and retrieves the results through buffer R. However, it is more efficient to leave large intermediate results on the slave node in a buffer I, than to send these results to the master. The users of the intermediate results, which are selected by the master, can retrieve the messages directly from I.

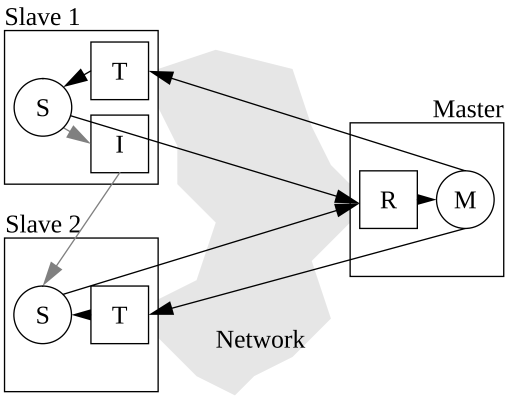

Figure 1: Addressing by receiver

Receives from remote buffers gives programmers more control over the message transfer. In the traditional communication model, messages are transferred over the network as part of the send, and, if necessary, they are buffered at the receiver side until the receiving task reads them. Sending large messages to receivers that are not ready to use the data can result in buffer overflow and ill-timed retransmissions. The problem is that the runtime system cannot know when the receiver needs the data. Having the receiver fetch the data when it is needed is an elegant and effective solution. Figure 2 again shows an example of a master task distributing work to slave tasks. However, slaves leave the results in local buffers where they can be retrieved by the master.

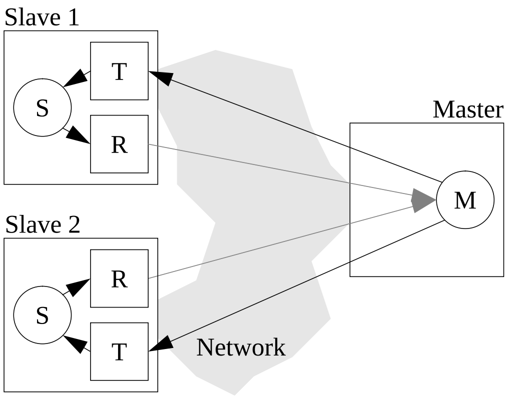

Figure 2: Flow control by receiver

Even though the symmetrical communication interface does not match the hardware as closely as the traditional asymmetrical interface, it gives the programmer more control over over how and when data is transferred over the network. Receives from remote buffers can of course be implemented at the application level: the receiving task can request the data from the sending task. This, however, will typically result in more complicated programs, and will also be less efficient since the request for the message has to propagate up to the application level, instead of being handled in the runtime system.

2.2. Message description

Send and receive primitives have to specify where the message should be retrieved or placed. In a language such as C, the simplest representation consists of a pointer-length pair that describes a contiguous byte area. This ‘‘immediate’’ representation can be improved both in terms of performance and ease of use.

The immediate representation can introduce copying overhead both for sending and receiving. For example, immediate receives from a local buffer require that the data is copied if the message is available before the receive is issued. Copying of the message or extra synchronization can be avoided by using ‘buffered’ messages. Buffered messages are built and consumed in areas that are managed jointly by the application and the runtime system. With a buffered send, the application builds the message in an area that was allocated specifically for that purpose, and the space is freed by the runtime system after the send is completed:

```
IMMEDIATE
    send(.., pointer, length, ..)

BUFFERED
    area = allocate(..)
    /* build message in area */
           ..
    send(.., area, ..)
    /* area deallocated by
       system after send */
```

A buffered receive returns a pointer to the message in a system buffer. The space taken by the message has to be freed explicitly by the application after the message has been read:

```
IMMEDIATE
    receive(.., pointer, length, ..)

BUFFERED
    area = receive(..)
    /* system allocates area */
            ..
    /* message used in area */
            ..
    deallocate(area)
```

From the application’s perspective, the main difference between immediate and buffer communication primitives is that in the latter case, the system determines where message have to be built and consumed. Our experience with Nectar shows that both types of primitives are useful. In some cases, applications send and receive messages that contain data that is part of a larger, existing data structure. In that case, immediate messages are the most natural. In other cases, the message is generated or consumed ‘‘on the fly’’, and having the location of the message selected by the system is not a problem. In both cases, having to use the different type of message description can result in inefficiencies.

Applications usually operate on structured data, and using variable-sized byte areas for communication is tedious and error-prone. Allowing users to send and receive data items of user-defined types is more desirable. On heterogeneous multicomputers, using typed messages also allows the conversion between different data representations to be done in the runtime system, if so desired by the application. The Nectar interface does currently not support typed messages.

2.3. Behavior of communication primitives

The merits of synchronous and asynchronous primitives have been discussed at length in the literature [4, 1]. Based on our experience with applications we see a small number of useful usage patterns:

- **Blocking mode**: The send or receive returns when the operation has been finished. Blocking primitives are the easiest to use and typically introduce the least overhead.
- **Timeout mode**: The operation returns when the operation is finished, or when the timeout expires. Timeouts simplify the task of detecting programming errors, such as mismatched send-receive pairs, and system errors, such as non-functioning nodes. Timeouts are not always appropriate. For server-type tasks, for example, it can be valid to be blocked in a receive for an indeterminate time period.
- **Non-blocking mode**: Operation returns (aborts) if it cannot complete immediately because of a full or empty buffer. The application can continue processing, and retry the operation later. The non-blocking mode is similar to using a timeout of zero, but the expiration of the ‘‘timeout’’ does not correspond to an error, so it has different performance requirements.
- **Asynchronous calls**: The application and communication operation execute concurrently, if possible. This mode can increase concurrency, for example for remote receives, or for local operations that require the copying of data using DMA. The asynchronous behavior is orthogonal to the choice of mode (blocking, timeout, or non-blocking).

The above primitives are not all equally efficient. Both setting up a timeout to catch errors, and setting up an asynchronous operation can add overhead.

Two parameters are needed to fully specify the behavior of a call: a timeout to select the mode (blocking, timeout, or non-blocking), and a parameter to select synchronous or asynchronous behavior. Although this results in a large number of options, this interface is simple both because it is orthogonal, and because the options correspond directly to specific usage patterns.

Asynchronous operations do not make sense for all operations on all architectures. Immediate operations that require copying of the data, for example, can run asynchronously with the application on architectures that have DMA support, but they cannot if the copying has to be done by the processor. In the latter case, the system should simply execute the operation synchronously.

Synchronous operations directly return a result: a status (success, failure, or timeout) for sends, and a status and the message for receives. Asynchronous operations always return a synchronization variable, that can be used by the application to find out when the operations finishes, and what the result is.

The definition of asynchronous primitives is not strictly necessary: applications can implement their own asynchronous primitives using dynamic processes or threads [4]. We decided to provide asynchronous operations for two reasons. First, asynchronous operations are common enough that it makes sense to support them in the runtime system; this eliminates one possible source of programming errors. Second, in many cases asynchronous operations can be implemented more efficiently than by using a separate thread. These alternative methods will often be architecture specific, and might not even be directly accessible to applications.

2.4. Format of communication primitives

In summary, all communication primitives have the following format:

```
Communicate(send/receive,dest_port,message,
                          timeout, sync/async)
```

where all parameters can be selected independently:

send/receive	place/retrieve a message in/from a buffer.
dest_port	address of the buffer (Section 2.1).
message	specification of where the message should be retrieved or placed (Section 2.2).
timeout	does the operation block, use a timeout, or return if it cannot complete immediately (Section 2.3).
sync/async	does the operation run synchronously or asynchronously (Section 2.3).

Implementations often use different primitives for at least the send/receive and sync/async features.

2.5. Defining the end of an operation

A receive operation is finished when the message is available to the application, but several definitions are possible for sends:

1. *Strict definition*: A send is finished when the message has been placed in the destination buffer. This definition is similar to that of a receive, and it presents a clean programming model. The problem is that it takes at least two roundtrip times to finish a send: one for the actual message, and one for an acknowledgement.
2. *System definition*: A send is finished when the area that holds the message can be safely reused by the application. This definition is typically used in the distributed systems area. ‘‘System sends’’ typically return earlier than ‘strict sends’. Examples include sends on systems that copy messages to a system buffer, or that have a reliable hardware send. The system definition has several disadvantages: debugging is more difficult, since sends can return successfully and fail later, and the system has to

handle a more complex flow control problem, since messages can accumulate in the network and system.

Nectar uses the system definition for performance reasons.

3. Implementation on Nectar

Nectar [2] is a high-speed network that was designed specifically to support heterogeneous multicomputers. The Nectar net consists of fiber-optic pairs and crossbar switches, and nodes are connected to Nectar through communication processors (CABs). The CABs are responsible for protocol processing and they can also execute application code. Several transport protocols were implemented [3]. They include Nectar-native protocols (the unreliable datagram, reliable message, and request-response protocols), and the standard internet protocols.

The Nectar interface follows the format described in Section 2. It uses symmetric addressing, has buffered and immediate communication primitives, and offers the options for blocking and synchronization of Section 2.3. Message are untyped. The Nectar interface is available on both the host and CAB.

Sends to remote buffers are implemented using the reliable message protocol. Receives from remote buffers are implemented on top of the request-response protocol: the runtime system sends a request to the remote node and the (application) data forms the reply. Note that the number of network packets is the same in both cases: a request and data packet for a receive, compared to the data plus and an acknowledgement for a send. The remote receive is handled entirely by the CABs. Buffers and buffered messages are located in CAB memory where they can be accessed directly by both the network interface and application tasks, without copying. Immediate messages sent or received by a task on the host are copied by the CPU. Asynchronous operations are handled by the CAB, with minimal overhead for the application.

Besides the communication primitives outlined above, the Nectar interface supports several other primitives. These features were added because Nectarine is the lowest-level interface in the system, and we wanted to make as many of the low-level hardware and protocol features accessible to the programmer as possible. The additional features include:

- **Protocol selection**: Users can select a protocol for communication operations that go over the network. This allows applications to choose between reliable protocols, and unreliable protocols that have less overhead. All applications on Nectar currently select the reliable protocols.
- **Connections**: We added connections to Nectarine, so that traditional connection-based protocols such as TCP/IP could be integrated. None of the applications currently use connections.
- **Request-response protocol**: Nectarine gives programmers direct access to the request-response protocol. It allows programmers to implement client-server style interactions using two network packets, as opposed to four packets with two separate messages. The request-response primitives are used regularly, but less often than the basic communication primitives.
- **Multicast**: The Nectar net supports multicast in hardware. Applications can use the hardware multicast by creating groups of buffers, and using the group address as the destination of a send. Multicast works for all Nectar-native transport protocols. Almost all Nectar applications use multicast. Typical uses include initialization and maintaining global state in the application during the computation.

Besides communication, the Nectar interface also supports the management of tasks on hosts and CABs.

4. Summary

We described a communication interface that was designed specifically to support user-level distributed applications. The interface has symmetrical send and receive operations, and all communication operations have the same regular format: different features are specified by orthogonal parameters. Although these primitives do not correspond directly to the low-level primitives supported by the hardware, they can be implemented efficiently, and they give the user control over how the message transfer is done. This interface is used as the lowest-level communication interface for the Nectar multicomputer.

References

1. Gregory R. Andrews and Ronald A. Olsson. "The evolution of the SR language". *Journal of Distributed Computing 1* (1986), 133-149.

2. Emmanuel Arnould, Francois Bitz, Eric Cooper, H. T. Kung, Robert Sansom and Peter Steenkiste. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. Third International Conference on Architectural Support for Programming Languages and Operating Systems, ACM/IEEE, Boston, April, 1989, pp. 205-216.

3. Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol Implementation on the Nectar Communication Processor. SIGCOMM ’90 Symposium on Communications Architectures and Protocols, ACM, Philadelphia, September, 1990, pp. 135-143.

4. B. Liskov, M. Herlihy, and L. Gilbert. Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing. 13th Annual ACM Conference on Principles of Programming Languages, ACM, January, 1986, pp. 150-159.