MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Some Considerations for API Semantics of Data Transmission

Randy Osborne

TR94-20 December 1994

Abstract

This contribution discusses the data transmission aspects of API semantics for native ATM services. Specifically, this contribution: 1. clarifies the semantics of the send and receive primitives presented in the baseline document 94-0150R5. 2. presents an alternative model for data transmission that is appropriate for emerging applications with high bandwidth and low latency requirements. This alternative model is intended to complement the socket-like send-receive model in the baseline document. 3. describes API flow control mechanisms required for data transmission.

ATM Forum SAA API Subworking Group, contribution 94-1142

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories, Inc.; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories, Inc. All rights reserved.

Copyright © Mitsubishi Electric Research Laboratories, Inc., 1994
201 Broadway, Cambridge, Massachusetts 02139

1. First printing, November 10, 1994

1 Introduction

The SAA API subworking group is defining an API to present access to native ATM services such as signaling and connection-based data transmission. The present semantic description of the API in the baseline document 94-0150R5 [BaseDoc] appears to be modeled after BSD sockets. (Prior SAA API contributions make it clear that a socket-like model was the intent at an earlier point.) The popularity of the socket model for networking makes it an obvious and important choice for an API. This model is well understood in the networking community and many applications have been written for it.

However, a socket model is not ideal as the exclusive means for access to native ATM services. In particular, the socket-derived send&receive model for data transmission in [BaseDoc] introduces communication overhead that hinders certain applications e.g. real time computing and emerging cluster-based computing paradigms. In order to allow access to the full range of native ATM services it is important to consider other models for the API in addition to sockets.

This contribution concentrates on the data transmission aspects of the API, taking the socket-like model as adequate for signaling, and proposes a wider view than send&receive for data transmission. In particular, this contribution presents an alternative model for data transmission aspects of the API that is appropriate for emerging applications with high bandwidth and low latency requirements. This alternative model is intended to complement the socket-like send&receive model in the baseline document.

1.1 Summary of Document

Section 2 presents the data transmission primitives in the baseline document, illustrates that the semantics of these primitives are unclear and incomplete, and then clarifies these semantics based on a send&receive model. Section 3 presents an alternative model for data transmission intended to complement this send&receive model, followed by discussions on the limitations of the send&receive model and emerging applications precluded by these limitations. Finally, Section 4 follows up on flow control issues raised in Section 2 and discusses flow control mechanisms required for data transmission.

2 Clarification of Data Transmission Semantics

The data transmission primitives in the baseline document [BaseDoc], pertaining to the AAL interface, are underspecified. After some discussion, this section provides a clarification consistent with a socket-like model (as appears to have been the intent from earlier SAA API contributions).

2.1 Send primitive

The baseline document describes the send primitive as follows:

```
ATM_API_SEND_DATA(
        IN communication_endpoint,
        IN data,
        OUT byte_count
 )
```

The `byte_count` specifies the number of bytes sent.

It is not clear when `ATM_API_SEND_DATA()` returns: does it merely enqueue the data (or a pointer to the data), trusting the driver/network interface to send it later, and return; or does it wait until the data has been injected into the network? If the former, how does it know what `byte_count` to return (since the data may actually be sent by the network interface at some later point). If the latter, must it be synchronous – can there be an asynchronous indication/notification when the network interface actually sends the data? In short, is the primitive contracting for data transfer to the driver or data transfer to the network?

There's also flow control issues. An application could send more data than the driver/network interface can accept. With ABR traffic, the network capacity devoted to an application can change at any time. Thus the API must provide some sort of flow control feedback to the application.

For CBR traffic, some sort of overflow feedback such as just described is still required (though it may be slower to act, and hence mostly appropriate for occasional instances). In addition, it is useful to have underflow feedback: e.g. a notification when a outgoing CBR queue size with the driver/network interface drops below some low watermark.

QoS issues also arise. At the sender, QoS amounts to making sure the application can deliver data to the network at the contracted rate. This necessitates both overflow and underflow feedback mechanisms to the application.

Finally, there remains a number of other semantic issues: How to specify where the data comes from? (What does `IN data` mean?) Stream vs. message mode? Unreliable or reliable data delivery?

2.2 Receive primitive

The baseline document describes the receive primitive as follows:

```
ATM_API_RECEIVE_DATA(
        IN communication_endpoint,
        OUT data,
        OUT byte_count
 )
```

In the case of stream mode, the `byte_count` parameter is the number of bytes received.

As with its send analogue, it is not clear when `ATM_API_RECEIVE_DATA()` returns: does it return immediately with whatever data (or a pointer to data) that may be available; or does it block until some quantity of data is available? Must it be synchronous – can there be an asynchronous indication/notification when the network interface actually sends the data to the application? In short, is the primitive contracting for data transfer from the driver or data transfer from the network?

There's also flow control issues. What if an application doesn't do a receive call, or do receive calls quickly enough – what happens if driver/network interface buffers overflow? Although for ABR traffic it is possible to imagine such feedback affecting the network flow control and thus backing up to the source, the time constant and complexity involved suggests that this is more easily handled as a destination application responsibility. Thus applications may be charged with performing receive calls at regular intervals. Still, there must be a mechanism for the driver/network interface to notify the application of such need. This is true for both ABR and CBR traffic.

QoS issues also arise. At the receiver QoS amounts to making sure the network interface/driver can deliver data to the application at the contracted rate. This necessitates an overflow feedback mechanism to the application.

Finally, a number of other semantic issues: How to specify where the data goes? (What does `OUT data` mean?) Stream vs. message mode? Unreliable or reliable data delivery? How is information about experienced congestion and possible data corruption conveyed to the application?

2.3 Clarified semantics

There are many positions to take with respect to the above issues. This section clarifies the above semantics in a way consistent with a socket-like model. We present very simple interfaces: we provide only overflow protection and assume unreliable data delivery.

2.3.1 Send

We separate the "`data`" field into a pointer to an application buffer and a count of the number of bytes to send starting from this buffer point.

```
ATM_API_SEND_DATA(
        IN communication_endpoint,
        IN buff,
        IN len,
        OUT byte_count
 )
```

`ATM_API_SEND_DATA()` attempts to transfer `len` bytes starting from the pointer `buff` to the connection indicated by `communication_endpoint`. The call blocks until completing this attempt at which time it returns with `byte_count` equal to the number bytes (starting from `buff`) that were transferred. A value of -1 for `byte_count` indicates that an error occurred and an undetermined number of bytes were transferred.

Remarks: The point at which the data is actually transmitted through the connection is implementation dependent. An implementation may simply transfer the data to a queue in the driver/network interface and return from the call. In this case `byte_count` indicates the number of bytes transferred across the API and not necessarily out on the network. A queue overflow may result in `byte_count` $<$ `len`. Thus note that the data delivery is unreliable. An error may include an overflowed queue. The application effects flow control by checking the `byte_count` after attempting a send.

2.3.2 Receive

```
ATM_API_RECEIVE_DATA(
        IN communication_endpoint,
        IN buff,
        IN len,
        OUT byte_count
 )
```

`ATM_API_RECEIVE_DATA()` attempts to transfer `len` bytes from the the connection indicated by `communication_endpoint` to a buffer starting from the pointer `buff`. The call blocks until completing this attempt at which time it returns with `byte_count` equal to the number of bytes transferred. A value of -1 for `byte_count` indicates that an error occurred.

Remarks: An receive queue overflow may lead to `byte_count` < `len`. The handling of a receive queue overflow in the driver/network interface is implementation dependent. We assume there is some mechanism, such as a notification/interrupt/signal/callback to the application, or even as simple as dropping all further incoming data while the queue is full and setting an error condition.

Whether or not data is actually copied is implementation dependent. An implementation may buffer data on memory pages and then effect a receive by mapping such a page into application space.

2.3.3 Further Remarks

Many semantic details are still missing from the above. For a description of the details associated with the format of the data units see 94-0813 [Singer]. We assume an indication/notification mechanism for conveying experienced congestion and possible data corruption to the application. Most of the details beyond this are implementation dependent. Finally, note that another layer can be added to the above API for reliable data transport.

3 An Alternative Model for Data Transmission

An API aspiring to give access to native ATM services should allow flexibility to meet requirements for high bandwidth **and** low overhead communication. Section 3.1 presents an alternative model for data transmission intended to complement the send&receive model discussed in Section 2. The sections following provide motivation and justification: Section 3.2 describes limitations of the send&receive model, Section 3.3 presents emerging applications precluded by these limitations, and Section 3.4 gives requirements of such applications.

3.1 The Model

We propose adding the following simple model as an optional extension of the API described in the baseline document 94-0150R5.

3.1.1 Overview

The data transmission model consists of the following major elements:

1. receive via notification. That is, message reception is asynchronous. There is no need to perform an explicit call to receive message data.
2. minimal data copies (preferably zero, for direct data delivery). Message data is directly delivered to the application space where possible and otherwise the application notified via an interrupt, signal, or callback that a message has arrived.
3. a mechanism to add application defined control information to messages at the source.
4. a mechanism to define the interpretation of the message control information upon message reception (we call this a receive handler). This receive handler should execute at the lowest possible level in the system. That is, an (expensive) context switch to application space on every message arrival to execute the receive handler should be avoided where possible. Section 3.1.3 discusses possible implementations.

Whereas elements (1) and (2) set the spirit of the model, elements (3) and (4) are the real semantic core (note that (4) subsumes (1)). Together they present a capability for low level, application-specific asynchronous actions on message delivery. Asynchronous delivery avoids the latency problem of send&receive. The general purpose handler allows a fast way to determine how to process the message, avoiding the data placement causality problem in Section 3.2 and providing sufficient flexibility to meet the requirements in Section 3.4. As an example of the power of this approach, elements (3) and (4) can be used to build a send&receive model like in Section 2.

Various means already exist to extend the socket-like send&receive model for asynchronous delivery e.g. signals and select calls in UNIX and callbacks in Windows Sockets. However, these means generally fail to achieve element (2) since the message data is stored in an intermediate location before the notification is issued. (See Section 3.2 for more discussion of this issue.) Some optimizations exist to reduce data copies, such as ADCs [Druschel], whereby incoming messages are stored directly in application space at locations indicated in circular queues shared between the application and network interface. However, the direct placement of data is limited to locations pre-contained in the free buffer queue. Finally, elements (3) and (4) are more powerful than the Winsock callbacks: the message filter operation is an arbitrary handler rather than just a read mask and the handlers do not require context switching to application space to execute unlike callbacks.

3.1.2 Example primitives

For concreteness we cast our model in the form of the following example API primitives:

```
ATM_API_SEND(
        IN communication_endpoint,
        IN controlp,
        IN control_len,
        IN buff,
        IN data_len,
        OUT byte_count
 )
```

`ATM_API_SEND()` attempts to transfer a message to the connection indicated by `communication_endpoint`. The message consists of `control_len` bytes of control information starting from the `controlp` pointer followed by `data_len` bytes of data starting from the pointer `buff`. The call blocks until completing this attempt at which time it returns with `byte_count` equal to the number of bytes transferred (starting from `controlp` for the first `control_len` bytes and then starting at `buff` for the remainder). A value of -1 for `byte_count` indicates that an error occurred.

```
ATM_INSTALL_RECEIVE_HANDLER(
        IN communication_endpoint,
        IN control_block_interpreter,
        OUT old_handler,
        OUT result
 )
```

`ATM_INSTALL_RECEIVE_HANDLER()` installs the one argument procedure `control_block_interpreter` (given as a pointer) as the receive handler to be executed on receipt of a message for the connection indicated by `communication_endpoint`. This procedure has the signature `control_block_interpreter(IN msg_ctl_blk)` where `msg_ctl_blk` is a pointer to the control information of an arriving message. This procedure should decode the control portion of an arriving message

and must work in concert with the control information structure provided by the sender. After installation of the handler, the call returns with `old_handler` pointing to the old receive handler (which may be NULL) and result set to a value ≥ 0 on success and -1 on failure (i.e. if an error occurred).

3.1.3 Implementation Issues

Section 3.1.2 gives only a general semantic description. In practice, to ensure both low overhead and protection of other processes from errant receive handlers, an implementation will probably not follow this description literally. As mentioned in requirement (4) in Section 3.1.1, it is imperative for performance that the receive handler run as the lowest possible level of the system. To accommodate this goal, some receive handler operations representing common operations may be predefined and implemented directly by hardware (as in the hybrid deposit model [Osborne]) to avoid interrupts to the kernel on every message arrival. In the absence of hardware support, an implementation may provide predefined fast interrupt processing routines (as in Active Messages [vonEickII]) in the kernel for common control operations. (Section 3.4 gives more details on Active Messages.) Although the suggested semantics are such that all operations dispatch through the `control_block_interpreter` procedure, an implementation will likely dispatch to any predefined operations directly, leaving the `control_block_procedure` for the interpretation of any operations not predefined by the system.

For example, an implementation may perform read and write operations using predefined hardware or kernel operations. A message with control information specifying one of these predefined operations would be handled directly by the hardware or kernel respectively without notifying the application.

A message with control information specifying some non-predefined operation will invoke a specified receiver handler procedure, executed in application space. If the application is not already running, this context switch will be expensive. Consequently, an implementation may elect to queue some messages involving such unsupported operations until that application is scheduled. Either a send&receive model could be used for queueing such messages or a ADC-like approach [Druschel]. While such message queueing reduces the overhead of message processing, it increases the latency. Thus it will be important for the sender to include sufficient information in the control portion of the message so that the destination can quickly determine (perhaps using a predefined operation) whether to immediately context switch to the appropriate application space and execute the receive handler or enqueue the message for processing when that application is scheduled.

We leave for future consideration the address space mapping and protection issues associated with direct delivery of messages to an application. [Thekkath] and [Osborne] give two methods for dealing with these issues.

Finally, section 4 discusses some of the ramifications various implementations have for flow control.

3.2 Limitations of the Send&Receive Model

The discussion in Sections 2.1 and 2.2 illuminates many issues in data transmission. A send&receive model, such as given in Section 2.3, while simple and familiar, is only one way of addressing these issues. Such a model embeds a particular choice of operations and costs that may be acceptable for some applications but not preferable for others.

There are two basic limitations with a send&receive model. The first is that data is not delivered until the application explicitly performs a receive operation. Such polling gives rise to a tradeoff of latency vs overhead, which is unsatisfactory for applications with low latency communication requirements. The second, which follows as a consequence of the first, is that data must be stored somewhere, independent of its intended use, upon arriving at a destination until the application performs a receive operation and

processes the data. Unfortunately, the fact that the application may merely want the data to be stored at some location (in application space) cannot be known until after the receive operation, at which time the data has already been stored somewhere else.

There are three solutions to this causality-induced data placement problem. The first solution is to copy the data. Simple implementations of the send&receive semantics in Section 2.3 adopt this solution by buffering data within the API implementation layers (i.e. driver) and copying it to an application specified buffer on a receive call. (Data may also be buffered within the API implementation layers at the sender.) However, it is well known that the main impediment to high bandwidth host network implementations is excessive data copying due to buffering. The second solution avoids copying by mapping the data. High performance implementations usually adopt this solution: the receive call causes the initial data locations to be mapped to an application specified buffer. While this eliminates a data copy, it has two consequences. First, to effect this mapping efficiently, the units of mapping are constrained to the relatively large sizes of pages. This puts a preference on the exchange of large data units in order to efficiently use the buffer space on a page and biases the structure of an application towards such large buffers. Second, the copying is replaced by a page mapping operation. Although page mapping is significantly cheaper than copying for a page of data, page mapping is nevertheless an operating system operation that is not cheap. This implies that the data transfer must be a certain size to amortize the cost of the mapping. (Small data transfers could still be done using copying, but that requires knowing in advance the size of the transfer.) The third solution is to change the receive semantics and have the receive call return a pointer to some application location in which the data is stored. This works fine so long as the application does not require the data in a particular location. Otherwise, the data must be copied or mapped, incurring the drawbacks of the first two solutions.

In the first two solutions small messages are proportionally much more costly than large messages. The third solution is not favorable for small messages either since small messages are very likely to be required to be stored at specific locations within an application, requiring copying or mapping. In addition, the rapid response messages for which the send&receive model already yields poor latency are typically small messages. Therefore, although the send&receive model can be optimized for large data, high bandwidth transmission, it is not suited for rapid exchange of small amounts of data, making it a bad choice for some applications.

3.3 Motivation: Small data transfers and Low overhead communication

While high bandwidth bulk data communication is obviously important, there is an increasing set of applications requiring small data transfers and low overhead communication. By low overhead we mean both low latency and low impact on host performance (for message processing).

Cluster-based computing — the networking of commodity processing nodes for high performance computing — is an emerging paradigm. The growth in the use of first generation cluster computing using Ethernet and such software tools as PVM [Geist] has been phenomenal. Latency, however, poses a fundamental limit to the effectiveness and applicability of cluster-based computing. There is no fundamental reason why ATM networks cannot provide low enough latency to provide the nucleus for future cluster-based computing. The raw network has little delay: interfaces can do message demultiplexing in hardware and switching can be fast. Some commercial switches right now get latencies of 2 to 3 cell times. However, current commercial network interfaces with send&receive APIs get latencies in the order of 250 microseconds [Osborne] and (much) worse. It is possible to do much better with alternative models. Various researchers have used standard commercial ATM interfaces to get application to application latencies below 30 microseconds [Thekkath, Osborne, vonEickI].

There is little doubt that cluster-based computing will be an important computing paradigm. Some go as far as to say that cluster-based computing will be the dominate computing model in the future

[Patterson]. And it isn't so much what one can do today with cluster-based computing but what one can do in the future. Low overhead communication will allow today's distributed computing applications to be restructured for better performance at lower cost. See [Thekkath] for one example.

Low overhead communication is also important for real-time computing, such as in industrial control systems.

3.4 Requirements for high bandwidth and low overhead communication

High bandwidth communication requires minimal copying of data. The objective is a "zero copy" model in which data is transmitted directly from sender application memory to destination application memory without any intermediate copies (by endstation drivers or operating system software). This direct delivery is like end-to-end DMA.

Low overhead communication also requires minimal copying of data. Other requirements arise from the manner in which low overhead communication is used. We identify three:

1. fast notification of message arrival – In this case an application needs to know as quickly as possible when a particular message arrives. This need arises, for example, when the message is a synchronization reply (e.g. lock grant) or other result (e.g. a read result from a remote node) for which the application is waiting.
2. fast reply to message arrival – In this case (really the remote analogue of the first) an application needs the result of a query or a remote node as quickly as possible. The round trip latency is important for many messages such as for reading a data value from a remote node or acquiring a lock on a remote node for which an application waits. Hence a fast reply to an incoming message is required.
3. random addressing – Messages requiring rapid response are typically small in size. Furthermore, applications with fine grained sharing and/or irregular data structures may transfer data in small messages. To avoid data copying we would like to have the ability to place data directly into the destination memory where the application expects it. This requires the ability to specify an arbitrary address, rather than just a linearly increasing address as in adding a packet to a receive queue.

An important aspect in realizing (3) is adding some sender supplied information (an address in this case) to a message. In fact, all three cases above require some sort of control information in addition to the data in a message. Active Messages [vonEickII] is a technique that exploits this observation by interpreting the control part as carrying the name of an interrupt handler to be executed at the destination. This interrupt handler may process additional arguments in the control part and then decide what to do with the message: whether to store the data or reply with some value. The hybrid deposit model [Osborne] relies on hardware support from the network interface to directly execute the most common and simple types of control (such as pure data delivery, as in a write) and falls back on Active Messages for less common control messages.

A send&receive model is not adequate to meet the above requirements. First, operating system interaction for data transfer and mapping ensures latency is large. Second, interpretation of a message's contents and hence reaction to a message cannot begin until the application performs an explicit receive call. Such a synchronous approach for asynchronous message arrivals gives unacceptable latencies.

4 Flow control

4.1 Recommendation

The API should include mechanisms on the sending and receiving interfaces for feedback of sender and receiver overflow respectively. Multiple methods are possible for the sender overflow feedback, including a mandated check of the sending status. However, the receiver overflow feedback must be in the form of a notification or callback in case the application is not scheduled.

The API may optionally include mechanisms to report underflow at the sender and receiver interfaces respectively. The sender underflow feedback must be in the form of a notification or callback in case the application is not scheduled.

4.2 Discussion and Rationale

As discussed in Section 2 the API must deal with flow control. The fundamental issues are:

- sender overflow – The network may not be able to send data as fast as the application can transfer data via a send call. Thus driver/network interface queues may overflow.
- receiver overflow – The application may not be able to retrieve (or use data) as fast as the network may receive it. Thus driver/ network interface queues may overflow.

Sender overflow is an vital issue for ABR connections: the network capacity devoted to such a connection can change at any time. Thus no amount of sender rate preplanning or queue size can prevent overflow. Furthermore, the ABR connection capacity can change rapidly. Consequently, it is essential that there exist a fast reacting feedback mechanism for flow control. (Of course, a particular implementation is free to simply drop data while an overflow condition exists, but this is not a satisfactory solution for all applications.) Fundamental ways to accomplish this feedback are by blocking (e.g. on a send call), polling (either explicitly or based on some value returned by a send call), or interrupt/notification/callback to the application.

Sender overflow is less of an issue for CBR connections since the guaranteed network capacity allows preplanning on the part of the application. An important distinction is that sender overflow with CBR connections is a result of mismanagement in the application or interference from other applications or the operating system. By making driver/network interface queues large enough this can be made sufficiently rare that a slow reacting feedback mechanism will suffice. Nevertheless, it is convenient for an application to have some way to determine remaining queue capacity; this can be used as a flow control mechanism.

The send&receive model has built-in flow control: send could just refuse to accept data.

One possible implementation of the data send in Section 3.1.2 to achieve minimal copies is a "non-buffered" send in which the sender queues data for transmission and the network interface transfers the data directly to the network. In this case there has to be some mechanism for checking if the driver/interface has accepted data for later transmission. This mechanism can be used for flow control.

Receiver overflow is also a vital issue for ABR connections: the network could deliver data at any point; moreover, the application may not be scheduled. Unless the application is guaranteed to attempt to receive data at regular intervals (e.g. via a timer) and the connection PCR is set appropriately, it is possible that receiver overflow occurs. Receiver overflow is also a vital issue for CBR connections unless the application can be guaranteed to be scheduled regularly and consume data.

The send&receive model needs an interrupt/notification/callback mechanism for such receiver overflow. Similarly there must be some flow control mechanism for "non-buffered" receives in which data is directly delivered to the application. Whether or not buffering is being done in kernel or application space, the buffers could fill up. A high water mark callback or notification is required.

Two additional issues important for guaranteed connection rates (e.g. CBR) are:

- sender underflow – There may not be sufficient data available for the network to send. For CBR connections, sender underflow may result in a violation of the guaranteed rate to the destination application. A similar issue can arise with ABR connections with non-zero MCR.
- receiver underflow – There may not be sufficient data available for the application. For CBR connections this condition might indicate a violation of the guaranteed rate and should be reported.

For CBR connections the send side must have a low watermark callback or notification. A receive side low watermark could be useful but does not seem essential. Even with MCR=0, ABR connections could benefit from a send side underflow: such a notification could alert an application to the opportunity to send more data.

5 References

BaseDoc - "Native ATM API Service Description Draft Specification", ATM Forum SAA API Subworking Group document 94-0150R5, Sept 1994

Druschel - "Experiences with a High-Speed Network Adaptor: A Software Perspective", Peter Druschel, Larry Peterson, and Bruce Davie, SIGCOMM 94

Geist - "PVM: Parallel Virtual Machine: A Users Guide and Tutorial for Network Parallel Computing", A. Geist el al, MIT Press, 1994

Osborne - "A Hybrid Deposit Model for Low Overhead Communication in High Speed LANs", R. Osborne, Proc. of IFIP 4th International Workshop on Protocols for High-speed Networks, August 1994

Patterson - "A Case for Networks of Workstations (NOW)", D. Patterson, First Networks of Workstations Workshop, October 1994

Singer - "ATM API Comments and Expanded Text", David Singer, ATM Forum contribution 94-0813, Sept 1994

Sunderam - "The PVM Concurrent Computing System: Evolution, Experience, and Trends", V. Sunderam, A. Geist, J. Dongarra, and R. Manchek, Parallel Computing, Vol 20(4), 1993

Thekkath - "Separating Data and Control Transfer in Distributed Operating Systems", C. Thekkath, H. Levy, and E. Lazowska, Sixth Int'l Conference in Architectural Support for Programming Languages and Operating Systems, October 1994

vonEickI - "Low-Latency Communication over ATM Networks using Active Messages", T. von Eicken et al, Proc. of Hot Intercomnects II, August 1994

vonEickII - "Active Messages: A Mechanism for Integrated Communication and Computation", T. Von Eicken et al, Intl Symposium on Computer Architecture, May 1992