Appendices for Data Plane Section of API Baseline Document

R. Osborne

TR95-11 December 1995

Abstract

This contribution proposes text for two Appendices of the API Baseline document 95-0008, covering aspects of the Data Plane layer. The text for the first Appendix covers pragmatics of data source and destination. The text for the second Appendix describes the optional extension to the messaging receive primitive mentioned in Section 3.3.2.3 of 95-0008.

1. First printing, April 10, 1995

Appendix A: Pragmatics of Data Send and Receive

Fundamental issues in sending data are:

1. identifying the location for outgoing data,
2. determining when the source location(s) may be reused, and
3. recognizing/handling sender overflow.

The most common way to specify the data source is to provide the location of a buffer (e.g. via a pointer) and amount of data to send starting from that location. Alternatively, the source data may be specified as an immediate operand to the send operation, i.e., the data is contained on the stack or in a register.

The source buffer cannot be reused until either the data has been deemed sent (or the attempt aborted) or the data has been copied to another buffer. Thus returning from a send operation does not necessarily mean the data has actually been sent. Depending on the communication protocol, (in the case of AAL5, depending on assured vs. unassured modes), the data might not be deemed sent until it is known to be successfully received at the destination. Typically, a send operation blocks until the source data is copied or mapped to an intermediate buffer (e.g. in the operating system/network driver). An implementation could also block until the data is deemed sent, but since this incurrs latency waiting for the network this usually has poor performance. Yet other possibilities are to poll until the reuse is indicated or to asynchronously interrupt/notify/callback the application to permit buffer reuse.

Sender overflow may occur if the network is not able to send data as fast as an application(s) can transfer data via send operations. In particular, intermediate buffers in the operating system/network driver may overflow. This is an important issue for ABR traffic: the network capacity devoted to such a connection can change at any time. Thus no amount of sender rate preplanning or buffer size can prevent overflow. Furthermore, the ABR connection capacity can change rapidly. Consequently, implementations may provide a feedback mechanism for indicating and controlling sender overflow. (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. It is often effective to tie the source generation rate to the availability of buffers for reuse.

Fundamental issues in receiving data are:

1. identifying the location for depositing incoming data,
2. indicating arrival of data to the application, and
3. recognizing/handling receiver overflow.

Typically, the operating system/network driver chooses a temporary intermediate location in operating system memory for the incoming data. However, it is also possible for the application to specify a location for the incoming data. For example, the application can indicate a memory range that can be transferred to the operating system/network driver as a buffer.

The application can determine the arrival of data via polling or blocking until the data arrives. Alternatively, the application can be informed of data arrival via an interrupt/notification/callback. In polling or blocking, once data has arrived it is transferred via copying or mapping to a location prespecified by the application in the receive operation. In the case of interrupt/notification/callback, the application can provide a location at the time of such an event to which the data it is transferred via copying or mapping.

Receiver overflow may occur if the application may not be able to retrieve (or use data) as fast as the network may receive it. As with sender overflow, intermediate buffers in the operating system/network driver may overflow. This is an important issue for ABR traffic: 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[1] is set appropriately, it is possible that receiver overflow occurs. Receiver overflow is also an issue for CBR connections unless the application can be guaranteed to be scheduled regularly and consume data. Since the application may not be scheduled as thus not able to consume and thereby replenish the intermediate buffers, there should be a interrupt/notification/callback mechanism for receiver overflow feedback.

For guaranteed connection rates (e.g. CBR), underflow may also be an issue. Sender underflow arises when there is insufficient 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. Likewise, receiver underflow arises when there may not be sufficient data available for the application.

Finally, it is very common to use operating system/network driver memory as a intermediate buffer for sending and receiving. Various work has explored eliminating the overhead of this approach by obtaining the source data directly from application memory and storing incoming network data directly into application memory. (For best results, this requires some support in the network interface architecture.) [Druschel] presents a good discussion of this approach. Closely related in spirit is work on Active Messages [vonEicken], and [Osborne] describes an ATM network interface architecture to support this approach.

Appendix B: Application-defined Control Information

Asynchronous receive, as described in Section 3.3.2.3, typically reduces the overhead of reception as compared to polling and blocking. However, the asynchronous receive described in that section notifies the application after the message data has already been received by the system and hence stored in some temporary intermediate location. To improve the performance of communication — both bandwidth and latency — it is desireable to eliminate the overhead of this intermediate copy as well. [Druschel] describes a means to support data transfer directly to the application without any operating system interaction or intermediate data copies. However, this approach unfortunately has poor resolution for the destination of data: data is stored in whatever pre-specified application buffer happens to be next in the free buffer queue. Consequently, the application may have to copy the data where it wants the data, negating some of the advantage of the direct data transfer. In many instances the sender knows or can know, e.g. by a request made by the destination, where the data should be deposited in the application memory at the destination. This Appendix describes an extension of the semantics in Section 3.3.2.3 which supports such direct placement of data at the destination.

B.1 Overview

The extension provides a means for the sender to include control information in a message that the destination can use to determine exactly where the message data should be deposited at the destination. The major elements of the extension are:

1. a mechanism to add application defined control information to messages at the source.

[1]Peak Cell rate

2. a mechanism to define the interpretation of the message control information upon message reception (a receive handler). This receive handler should execute at the lowest possible level in the system.

The general purpose handler allows a fast way to determine how to process the message without requiring intermediate storage of the full message (thus incurring copying overhead). [vonEicken] calls this idea Active Messages.

B.2 Example Primitives

The following constructs are illustrative examples.

```
ATM_send_cntl&data(
        IN endpoint_identifier,
        IN cntl_source,
        IN data_source,
        OUT sending_result
 )
```

This construct mirrors **ATM_send_data** (Section 3.3.1) in every way except that the message data sent consists of the control information described by `cntl_source` followed by the data described by `data_source`. `cntl_source` is a descriptor in the same format as `data_source`. The control and data fields are explicitly provided separately so that a given set of control information may be used in many messages without copying the control information to the front of each data block.

The following construct is highly system dependent.

```
ATM_install_receive_handler(
        IN endpoint_identifier,
        IN cntl_interpreter,
        OUT result
 )
```

`ATM_install_receive_handler()` installs the one argument procedure `cntl_interpreter` (given as a pointer) as the receive handler to be executed on receipt of a message for the connection indicated by `endpoint_identifier`. This procedure has the signature `cntl_interpreter(IN msg_cntl_blk)` where `msg_cntl_blk` is a pointer to the control information of an arriving message (which is at the head 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 `result` giving a status indication.

B.3 Implementation Issues

The previous section 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. 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 [vonEicken]) in the kernel for common control operations. Although the suggested semantics are such that all operations dispatch through the `cntl_interpreter` procedure, an implementation will likely dispatch to any predefined operations directly, leaving the `cntl_interpreter` 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.

There are 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.

References

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

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

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

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