

## **Interprocess Communications in the AN/BSY-2 Distributed Computer System: A Case Study**

David Andrews<sup>1</sup>, Paul Austin<sup>2</sup>, Peter Costello<sup>3</sup>, David LeVan<sup>3</sup>

<sup>1</sup>Department of Electrical Engineering and Computer Science  
University of Kansas  
dandrews@eecs.ukans.edu

<sup>2</sup>Xerox Corporation  
Rochester, NY

<sup>3</sup>Lockheed Martin Corporation  
EP-7 RM 121  
Electronics Park  
Liverpool, NY 13221

### **Abstract**

This paper presents a case study of the design and implementation of the interprocess communications facility developed for the AN/BSY-2 distributed computer system, the computer system for the Seawolf submarine. The interprocess communications facility was identified as a critical design challenge for the AN/BSY-2 system, as the system incorporated new component and network technology along with new run time system services as well as application programs. The requirements specified for the interprocess communications included aggressive performance, as well as functional capabilities that had not been previously fielded. The AN/BSY-2 computer system is comprised of over 100 processors interconnected in multiple fault tolerant fiber optic rings. First, a description of the AN/BSY-2 distributed architecture is presented. The message-passing semantics are then presented. A key feature of the IPC facility is its support for both synchronous and asynchronous communications based on logical addressing. Logical addressing within the AN/BSY-2 system supports point to point as well as group communications, and also supports the fault tolerant requirements of the system. The hardware developed to support fast real time messaging, and support fault tolerance is discussed. Finally, the low level semantics of a message transfer through the system is outlined.

## 1. Introduction

This paper presents a case study of the design and implementation of the interprocess communications facility developed for the AN/BSY-2 system, the data and signal processing computer system for the Seawolf [16][20] submarine. The AN/BSY-2 system, deployed in 1997, represented a major advance in embedded systems design, incorporating functional capabilities typically not included in a closed embedded system. A block diagram of the system is shown in Figure 1. The AN/BSY-2 system provides nearly 100 processing nodes interconnected on a fiber optic redundant ring network, high speed parallel and serial communications channels from external interfaces, communications channels to display consoles, standalone processors, and SCSI interfaces. In addition to meeting timeliness requirements, the system includes additional capabilities to support autonomous operation, dynamic system resource management, and fault detection and reconfiguration. The AN/BSY-2 system represents one of the first large distributed real time multiprocessing systems using the message passing paradigm, and is the largest real time system ever successfully fielded [17]. Due to its size, complexity, and mission criticality, the AN/BSY-2 system has also been used to study the effect of Ada coding styles on execution performance [18].

While the AN/BSY-2 system was being developed in the late 1980's, standards for several open source message passing interfaces for non-real time systems were being defined [2][3]. These interfaces sought to make use of the most attractive features of previously existing message passing systems. As an example, the MPI standard was strongly influenced by work at IBM T.J. Watson Research Center [4,5], Intel's NX/2 [6], Express [7], nCUBE's Vertex [8], p4[9], and PARAMACS [10]. Standard libraries based on the message passing paradigm have also been developed for specific applications [11]. The largest portion of the early work in developing open source message passing interfaces and standards were not specifically targeted at real time systems. More recently, a working group has been developing MPI/RT [14], a real time version of MPI. As implementation of message passing facilities became more efficient, its popularity as a scalable communications model continued to grow. The message passing model has now been adopted in a wide spectrum of application domains, including the newly emerging domain of micro-electrical-mechanical (MEM's) based next generation networked sensor systems [15].

The message passing facility was identified by the Navy at the beginning of the program as a critical component within the system due to aggressive performance and functional requirements that could not be met by any commercially available messaging software system. Developing the message passing facility and verifying that the system met all requirements was made more difficult as the AN/BSY-2 system was based on a custom hardware platform necessary to meet the overall system requirements. Further, the message passing facility would also be required by application programmers to support development of the 4 million source lines of Ada that would eventually run in the AN/BSY-2 system. Due to this need for simultaneous development of the hardware along with the run time system software, a scaled prototype system composed of commercially available components that functionally emulated the AN/BSY-2 system was specified for supporting functional development. The prototype system along, with

the run time software, could then be used for prototyping and functional integration of application programs. Although this approach mitigated risk by providing a convenient development platform, final integration and verification of requirements would be performed on the actual hardware once the system was fielded.

## **2. Message Passing Interface Overview**

One of the first system challenges was to define the requirements for an augmentation to the commercial adopted Ada multitasking run time executive to support operation in a distributed, real time environment. The augmentation was defined to support asynchronous operation and co-ordination of distributed program tasks, and control of common resources distributed throughout the system. The augmentation also included the fault detection and recovery requirements of the system. The augmentation, combined with additional system resource management software formed a system level middleware layer.

The interprocess communication (IPC) facility was defined within the augmentation to support fault detection, reconfiguration, and routing of messages across multiple communications channel media (serial links, parallel links, fiber optic links), within the context of real time timeliness constraints. These requirements, typical of an embedded real time system, provided challenges for defining the operational semantics of the IPC facility. Functionally, the IPC facility would support the dynamic bandwidth requirements of the system, promote platform independence, component based software development, and support system reconfiguration in the presence of faults. The functionality of the message passing facility also included support for development of the multiple applications running asynchronously throughout the system. With over 500 software engineers and 4 million Ada source lines, it was critical that the operating system provide a standard API for developers running on the prototype, and ultimately the fielded system. The asynchronous nature of the system required additional built in functionality for post-mortem analysis in several forms, including time stamping, and non-invasive message logging. The semantics of the API also provided access to additional state information that would not ultimately be accessed during deployment.

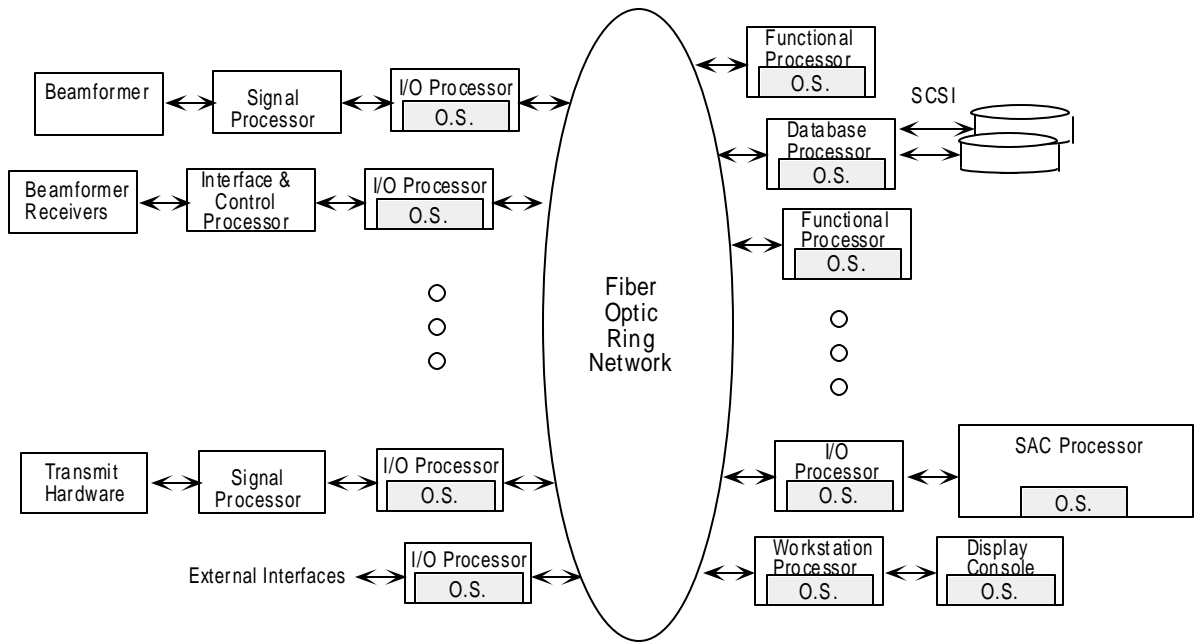


Figure 1. BSY-2 System Network Topology

The API provided a virtual channel abstraction, which hid network specific protocols, physical message routing, physical locations of disks, workstations, and processor nodes from the user. Figure 2 shows application programs forming virtual connections using the message passing interface. A program running on a particular processor as shown in Figure 1 may communicate with a program running on any other processor, requiring transmission of the message across multiple communications channels, each with their own specific device driver protocols. The message passing interface contains no information on these locations, and IPC guarantees delivery of the message. This separation of network specific information was necessary in order to support the requirements of the system outlined above. The additional functionality required in order to support a virtual programming model is discussed below.

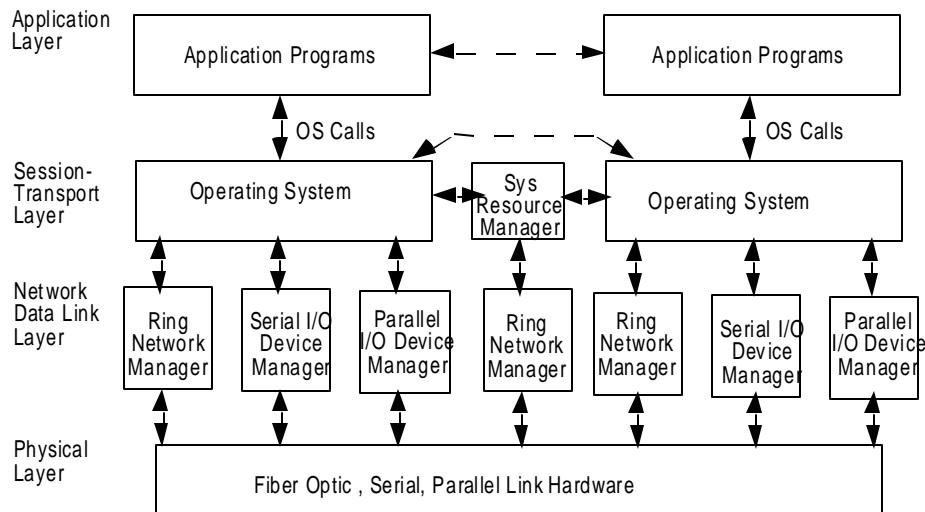


Figure 2. Abstract Programming Model

## 2.1 Logical Addressing

Programs open logical channels to send, and logical ports to receive messages. The logical channels and ports are opened by calls from the user program to the IPC facility. A program may open and close individual channels and ports at any time during program execution. Group opens are also provided to minimize the overhead of opening multiple channels and ports sequentially. The IPC facility registers each open request on the program's behalf with a global system resource manager. The global resource manager maintains a routing table that maps logical ID's to physical nodes within the system. Appropriate information is passed back from the global system resource manager to the IPC facility on all nodes requiring updates based on the addition or deletion of a node for a logical ID. This information is not required by the application program, and is maintained by the IPC facility transparent to the user.

Basing the channel and port addresses on logical identifiers provides several advantages to the system. First, group operations are easily enabled. Programs may be dynamically included or excluded in a logical group by opening or closing a port on the appropriate logical channel. Second and critically important, the logical channels support dynamic reconfiguration in the presence of faults. The system resource manager will relocate to a healthy node, programs unreachable due to faults or failures. Healthy programs may continue sending and receiving on the logical channel while the relocation is being performed. The IPC facility, network and system resource managers, provide this service transparently to the applications programs. Logical ID's also support application debugging and integration, requiring no change to application code for operation on either the test bed system, or real system. The base send API is shown below. Variants to the base call are also provided.

```

SEND( Channel_ID:  in CHANNEL_PTR_TYPE;
      MSG_SIZE:    in MSG_SIZE_TYPE;
      Buffer_Ptr:   in BUFFER_PTR_TYPE) return TX_PTR_TYPE;

```

The channel ID passed into the send is the logical ID. In addition to supporting the fault relocation requirements of the system, transfers based on logical IDs also provide support for platform independent point-to-point and group transfers. This versatility is shown in Figure 3 where the mapping of logical ID L1 as a port to receive messages in program 2 is transparent to program P1. The physical routing between any programs such as P1 to P2 in the example below, may occur over multiple communications media. If program P2 fails or terminates, the IPC facility will update all programs sending and receiving on the affected channels.

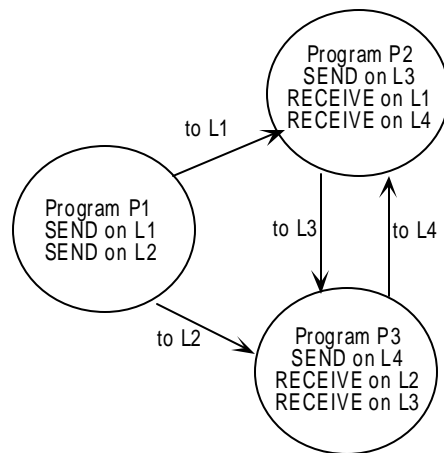


Figure 3. Logical IDs and Programs

## 2.2 Collective/Group Operations

Logical channels define the collective and group operations throughout the system. A logical ID will support multiple senders and a single receiver, multiple receivers and a single sender, or multiple senders and multiple receivers. This collective and group operation capability supports user level fault tolerance as well as reducing the processing time required by an application program to send the same message to multiple receivers. This capability also provides the system resource manager greater flexibility in routing without change to actual application programs.

## 2.3 Synchronous and Asynchronous Service

All message sends and receives are by default, asynchronous operations in the system. This default case allows user programs to initiate message transfers and continue processing while the operating system and network are transferring the message. The asynchronous transfer format minimizes the overhead penalty of sending and receiving

messages, allowing the user program to continue operations on user functions while periodically checking status. For applications that perform large numbers of multiple asynchronous sends, the user defines, and then passes through the send call a semaphore. As the status of each send is returned and updated, the IPC facility updates the semaphore. In this fashion, the application program need only check the single semaphore, and may elect to check the individual status for any number of completed events desired. This simple protocol reduces the status “polling” time for these types of applications.

Synchronous transmissions are synthesized by executing a separate suspend function after executing the send or receive. When suspend is executed, the operating system suspends further execution of the Ada task until a transfer complete status has been returned. When the task is suspended, the operating system initiates execution of the next program in the ready to run queue, as shown in Figure 4.

#### 2.4 Virtual Channel/Datagram Protocols

Both datagram and virtual channel protocols are supported by the IPC facility. Message transfers using the datagram protocol in the BSY-2 system are initiated by the IPC facility on the sending node. The IPC facility initiates the transfer on behalf of the sending application by sending the message to the network processor node. At this point no allocation of buffers or intermediate hardware resources has occurred. While the application program continues execution, the message is transferred throughout the network by dynamically allocating buffers and resources for the next intermediate destination. Multiple packets from the same message may exist simultaneously throughout the network and are dynamically allocated resources throughout the network.

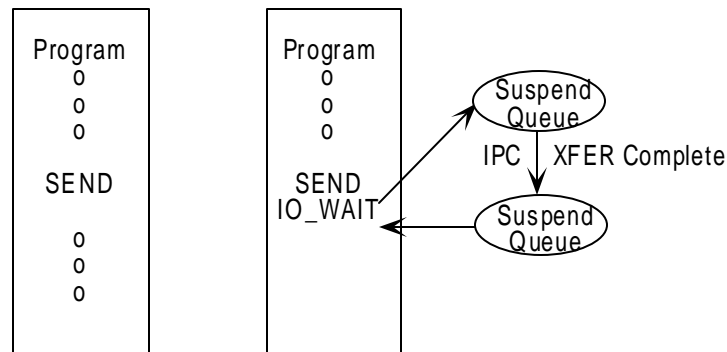


Figure 4. Asynchronous and Synchronous Message Transmissions

Virtual channel transfers initiate identically to datagram channels. The IPC facility initiates the transfer on behalf of the sending application by initiating transfer of the message to the network processor node. However, before the message is sent from the sending network processor, all intermediate buffers and resources are pre-allocated.

Virtual channels are used only in limited applications, and have the potential to cause starvation of intermediate network resources for large messages.

## **2.5 Message Segmentation**

IPC message sizes can range from a few bytes to the size of the largest buffer in the system (approx. 2+ Mbytes determined prior to system implementation). The network resources that limit the size of a single transfer are maintained transparently from the user applications. The IPC facility accepts messages of all sizes, and segments the messages based on the resources available for the specific communication hardware resources and protocols. The IPC facility maintains status and guarantees the transfer of all segments for each message.

## **2.6 Fault Tolerant Message Delivery**

The IPC facility must guarantee the integrity of the message during transfer. A hierarchical fault detection approach was implemented to provide overlapping detection of transfer errors. The hierarchy uses both built in hardware support, and additional software techniques for providing the error detection coverage. First, a polynomial based encoding of a Cyclical Redundancy Check (CRC) [14] is performed on each segment and segment executive header transferred between the application buffer and the network processor. Before each executive header is transferred, a simple checksum is included in the header. The final computed CRC is placed in a separate control message sent to the network processor. The network processor transfers the control message containing the CRC from the sending node to the receiving node within the message that initiates the transfer. During the subsequent processing of the receiving message, the IPC facility in the receiving node first checks the integrity of the header by verifying the checksum, and then initiates the transfer of the message body into a user buffer. A CRC is again computed on the segment and executive header during the transfer from the processor node into the application memory, and the IPC facility compares the CRC's to determine if any errors occurred. The ring network provides further built in coverage at the packet level. Any errors detected are reported back to the sending IPC facility for subsequent error processing.

Message failures can also occur due to other error conditions throughout the system, such as hardware faults or application program faults. The IPC facility accounts for these types of errors by setting a watchdog timer for each transfer. If a message is sent and the watchdog timer expires before a response is received, the IPC facility initiates error processing.

In the event a failure is detected, the IPC facility typically initiates a resend of the segment. The IPC facility provides a default maximum number of 3 retries per segment, per destination. In the event of a group transfer, individual segments may be retransmitted to specific destinations. For a subset of faults the IPC facility can resend and complete the transfer transparently to the application program. However, if the IPC facility is unable to successfully complete the transfer, an error status is returned to the application program, at which time the application program may elect to resend the message. If elected, the message is only resent to those destinations that reported

failures. Flow control is also implemented in order to balance aperiodic send and receive rates. If all resources (receive buffers) are currently being used by a receiving program on a given logical channel, the message transfer is suspended until resources are freed

### 3. IPC Hardware Support

The AN/BSY-2 computer system included built in hardware support for message passing, including parallel DMA channels, mailbox interrupts between the network processor and application processor, dual ported memory, and fast exception handling. Low level hardware support was also included for supporting fault location and detection, as well as fast efficient processing of the variable message sizes input from real time signal processing hardware.

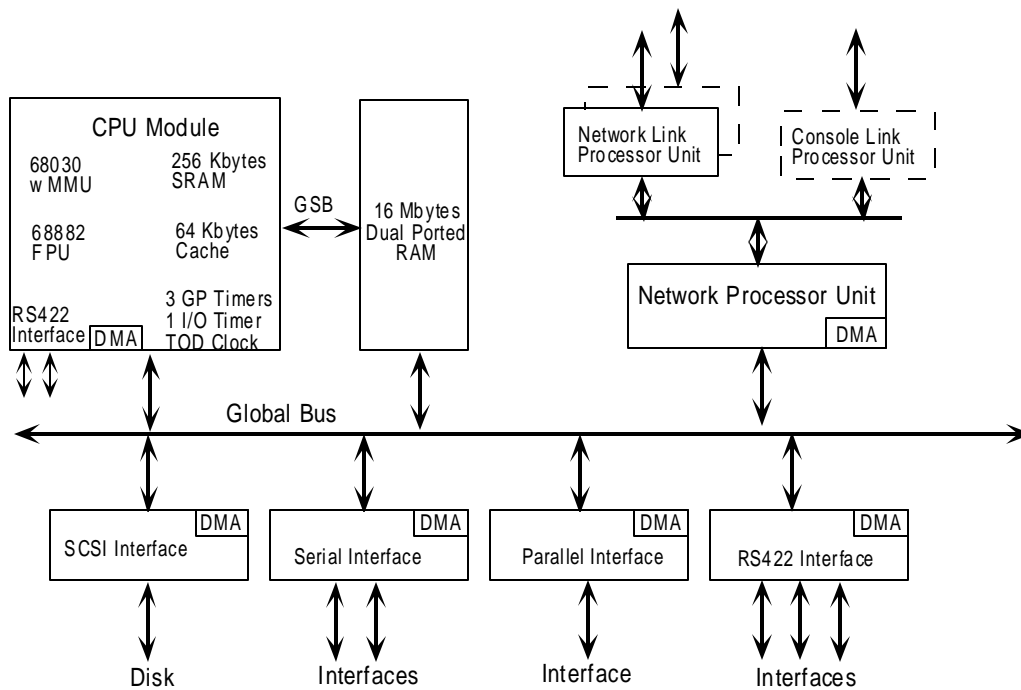


Figure 5. Processor Node Architecture

Figure 5 shows the built in low level hardware support for IPC. As shown in Figure 5, each communications channel (fiber optic, serial, and parallel) contains a dedicated DMA device to provide fast block transfers of data between user buffers and communications devices. The node architecture also includes a private bus between the CPU and dual port memory. This dual bus organization allows IPC to set up the DMA transfer across the global bus, and return control back to the application program. The application program can continue instruction execution out of the CPU cache, and may continue to access data from the dual port memory using the local bus. The dual bus organization and multiport memory minimizes busy waiting due to bus arbitration during the DMA message transfer. Mailbox interrupts are provided for communications channels to invoke IPC. Figure 6 shows an expanded view of the interface between the network

processor's multiport memory, and the global bus. Figure 6 shows the network processor node containing send and receive work queues, and dual port memory. The dual port memory is separated into two buffer pools, a send buffer pool and receive buffer pool.

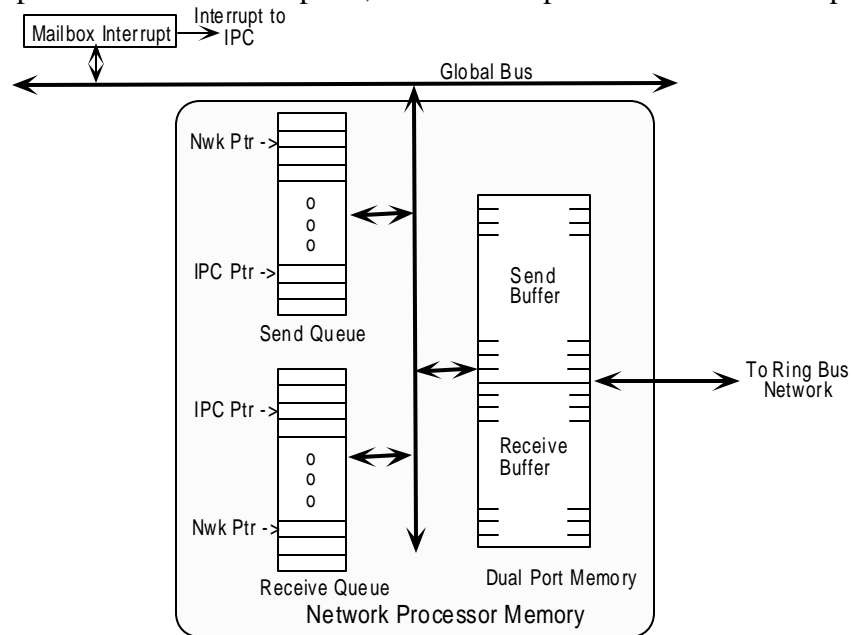


Figure 6. IPC-Network Interface

### Send/Receive Queues

IPC and the network communicate via Communications Control Messages (CCM) placed into the send and receive work queues. A CCM message is shown in Figure 7. For the send queue, IPC maintains the next free entry pointer, and the network processor maintains the next to service pointer. IPC places the next CCM entry into the queue, and sets the valid bit. Setting the valid bit causes an interrupt to the network processor. The network processor can service all sequential requests with a valid bit set. When the processor reaches an entry with the valid bit cleared, the top of the work queue has been reached, and the network processor terminates service. For the receive queue, the same protocol exists, but the roles of the network processor and IPC are reversed. The network processor places incoming requests into the queue, and writes into the IPC mailbox causing an interrupt to IPC. IPC then checks the receive queue for the next valid entry. Each work queue is maintained as a circular buffer. This protocol allows both the network processor and IPC to simultaneously operate out of the same queue. At this level, both the IPC facility and the network processor pull work requests from the queue on a first come first serve basis, implying no sorting or arranging of work requests based on priorities. Both the IPC facility and the network processor may then choose to insert requests into local scheduling queues based on priorities. This approach simplified the implementation of the interface work queues.

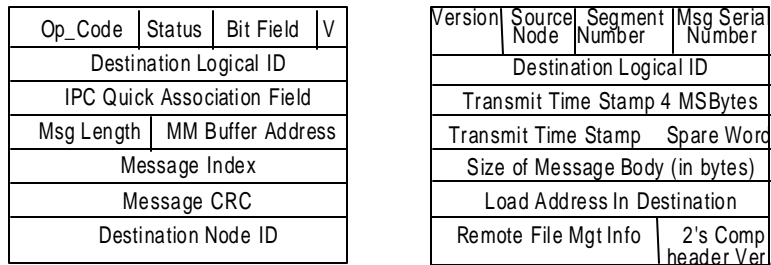


Figure 7. CCM and Executive Header Layout

### 3. IPC Implementation

The critical design features of the IPC facility included the organization of run time data structures to support the required functionality, efficient run time allocation/deallocation of these data structures, implementation of fast device drivers, and utilization of low level hardware resources supporting transfer of the data. Designing and implementing the data structures was a critical step in the overall design of the IPC facility. The real time requirements of the system placed hard timing constraints on allowable overhead processing time for allocation/deallocation of the data structures, and searching and updating the status in the data structures. Where possible, data structures were pre-allocated in pools to minimize run time overhead, organized to minimize search and update times, and quick association fields were included for accessing particular fields within more complicated and hierarchical structures. The organization of the data structures is discussed below.

#### Data Structure Design

Applications first execute an open\_channel command that registers the program with the system resource manager. Information is passed back to IPC from the system resource manager during the execution of the open\_channel command, causing the creation of a Communications Control Block (CCB) and Request Control Block (RCB) as shown in Figure 8 and 9, respectively.

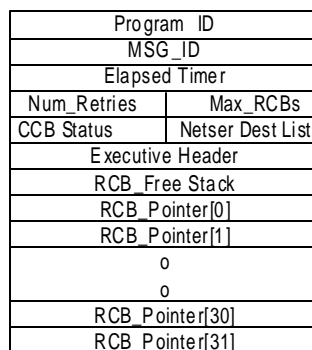


Figure 8. CCB Data Structure

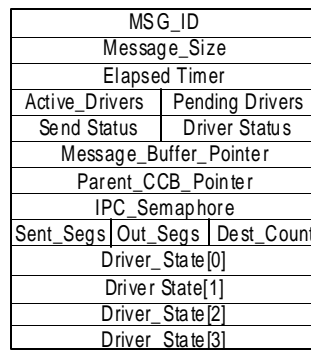


Figure 9. RCB Data Structure

The AN/BSY-2 system requirements included multi-destination sends, and multiple messages in flight on the same logical ID. The segments of a particular message are

routed independently through the network based on a dynamic routing protocol, and arrive at the destination in no particular order. This asynchrony requires the IPC facility to properly reconstruct the original message on the receiving node.

For each send operation performed by the application program, a free RCB is popped from the stack and associated with the single invocation. In the CCB shown above, 31 simultaneous messages can be sent asynchronously on the same logical channel. After a particular send has completed, the RCB associated with the send is freed by placing the pointer back on the free stack. The organization of an RCB is shown in Figure 9.

The RCB data structure keeps the updated status of the transfer. For each message sent over a particular communications medium, the RCB also keeps track of the number of segments sent in the message, and the status of each segment. The protocol defined for the system allows failure of an individual segment only a certain number of times. The RCB keeps a failed tally for each segment. When the number of maximum retries for failed segments are reached, the transmission to that particular destination is flagged as failed. In the case of a single destination, then the complete transfer is flagged as failed. However, for multiple destination messages, the message may have completed successfully to other destinations. The IPC facility maintains the status of each individual destination in the case of a multi-destination send, and makes this information available to the application programs through a return status. Application programs can invoke a re-send of the message if the status indicates a failed destination, causing the IPC facility to re-send only to those destinations that reported failure. The message is not resent to destinations that reported success. This information can also be used by the network resource manager in evaluating the system status.

#### **4. IPC Low Level Programming**

IPC transfers can be broken into three separate stages. The first stage is initiated by the IPC call from the application program to the operating system. In this first stage, the message is broken into segments, and the individual segments are sent over the appropriate communications channel. Figure 10 shows the transfer of a message to multiple destinations over the fiber optic ring. As shown in Figure 10, multiple segments of the same message can exist in the network processors multiport memory concurrently. Once an individual segment has been transferred from the user buffer into the network multiport memory, it may be transferred through the network independently from all other segments. In the case of multiple destination messages, the individual segments are transferred once from the applications send buffer to the multiport memory. The network performs multiple transfers of the same segment to each destination specified by IPC.

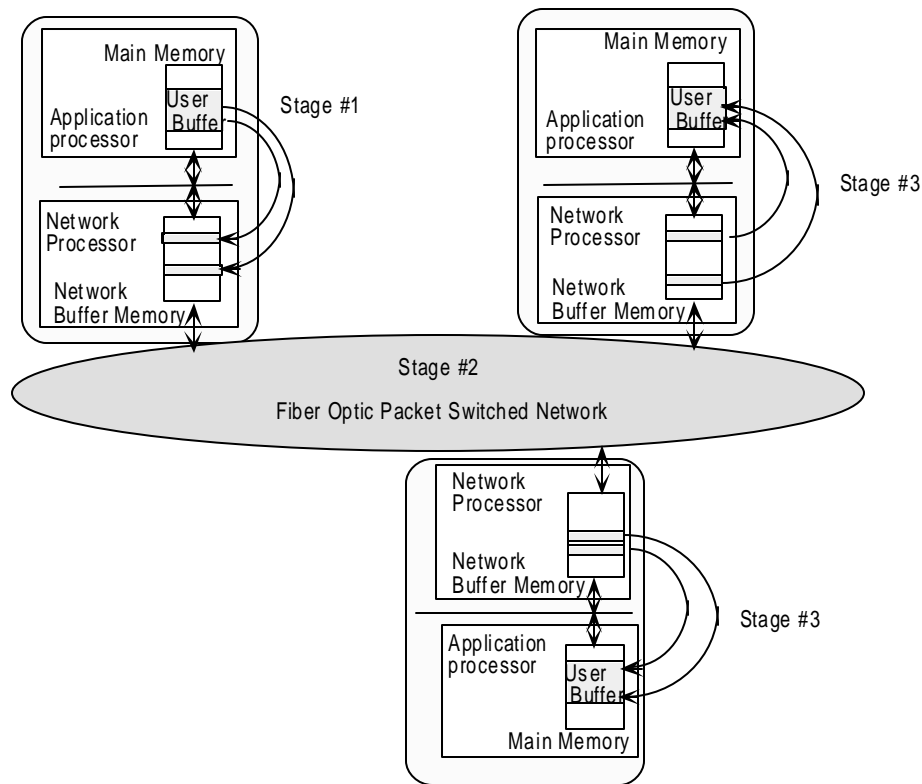


Figure 10. IPC Transfer Stages

The second stage is the actual transfer of the message from the sending network processors multiport memory into the receiving nodes multiport memory. The third stage is the transfer of the message segments from the receivers network processor into the receive buffer. As the segments arrive, the receiving IPC must reconstruct the message using the segment number information contained in an executive header included at the top of each segment.

Each of these three stages can operate asynchronously and concurrently for the multiple segments going to multiple destinations. IPC processes the transfer using a program initiated interrupt driven exception routine. Execution of IPC in an interrupt routine allows each transfer stage shown in Figure 10 to occur asynchronously, avoiding time consuming busy waiting, or constant status polling of all segment transfers and status updates. A simplified version of the state machine organization of the exception routine is shown in Figure 11.

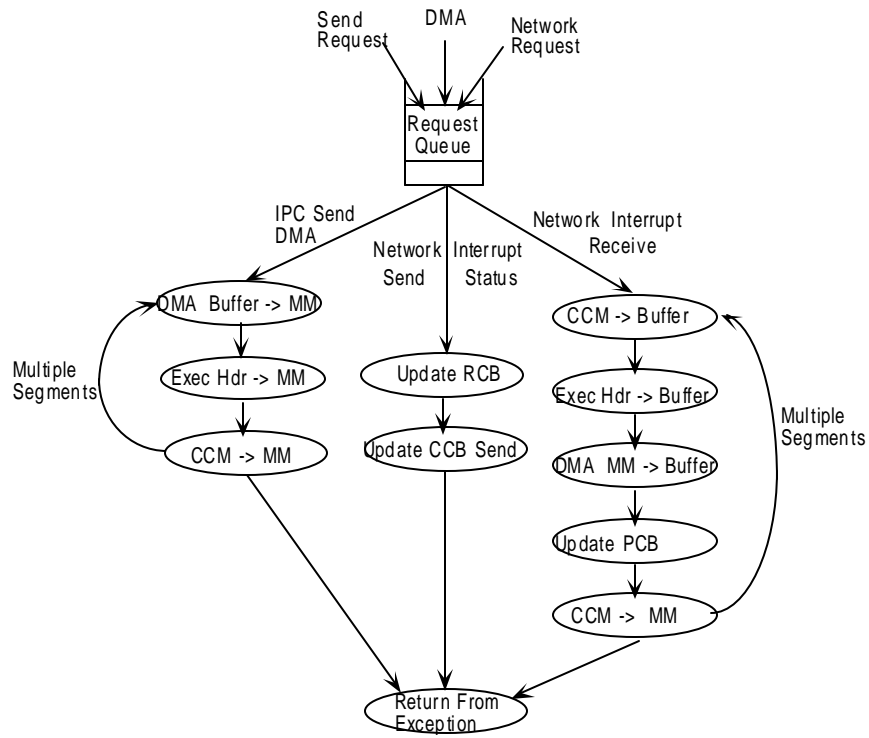


Figure 11. IPC Exception Routine Operation

If the IPC work queue is empty, or all requests have been serviced and IPC is idle waiting on return status, placing the entry into the work queue initiates the exception processing routine. If the work queue contains requests in progress, the request is queued.. Requests are generally handed in FIFO order, however certain higher priority requests can be inserted into the queue and processed relative to their priority. In the absence of any priority ordering, queued receive requests will take precedence over queued send requests. Control returns back to the application program after the request is placed in the queue. This implementation provides fast return back to the user program, minimizing the amount of time line taken by IPC for sending or receiving messages.

### Message Transfer Sequence

The protocol for transferring messages between IPC and the network processor is outlined in Figure 12. Initially, a segment from the send message buffer is transferred into the network multiport memory (1 in Figure 12). This transfer is accomplished using the network processor DMA unit shown in Figure 9. IPC sets up the transfer and returns control back to the user. The application program continues to execute while the DMA transfer takes place. Once the DMA transfer has completed, the DMA causes an interrupt that again kicks off the IPC exception routine. Next, an executive header is DMA'd into the top seven long word entries in the multiport memory buffer (2 in Figure 12). The executive header is shown in Figure 11. The executive header contains information required by the receiving IPC to reconstruct the message. The executive header does not

contain any information required by the network, and is passed through the network to the receiving IPC facility as part of the message body. The executive header forms a virtual communications link at the session/transport layer shown in Figure 2.

The CRC check is performed on the message and executive header. The IPC facility places the CRC into a CCM for that transfer, and writes the CCM into the send queue (3 in Figure 12). The IPC facility then updates the RCB status indicating that a segment was sent to a single destination and checks to see if more segments are available for transfer, or if more destinations are specified for the segment before leaving the exception routine. For each destination specified in the logical ID, a CCM is transferred into the send work queue. If more segments are available for transfer, the IPC facility sets up the DMA and exits the routine.

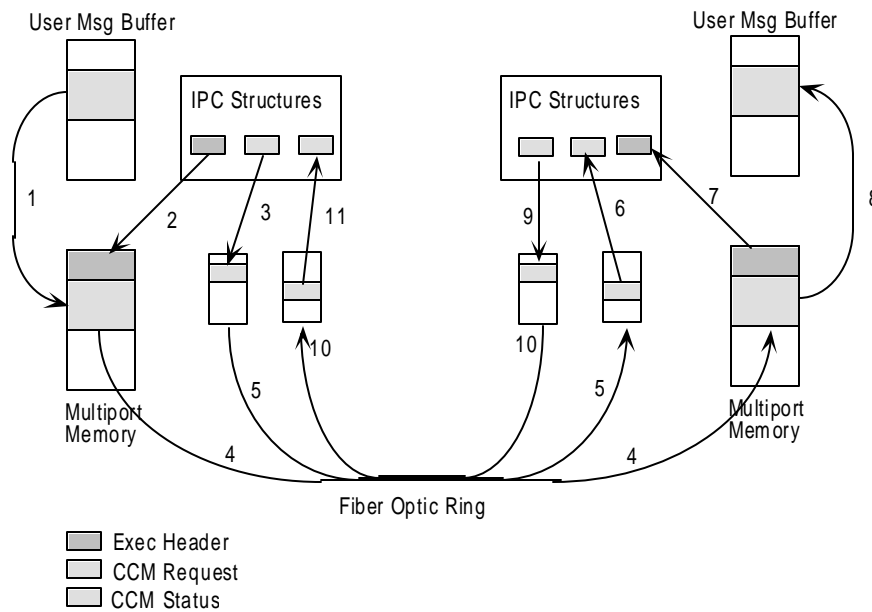


Figure 12. Message Transfer Protocol

The exception routine is next entered when the network processor writes into the IPC mailbox causing an interrupt. The IPC facility then transfers a CCM from the receive queue into the processor (6 in Figure 12) and checks the op\_code. If the op-code specified an incoming message, a DMA transfer of the executive header from multiport memory (7 in Figure 12) is initiated. The executive header must be transferred first, in order to determine the destination logical ID, and which segment of the message is pending in the multiport memory. The receive data structures are updated, and a DMA transfer is set up transferring the message from multiport memory into the receive buffer (8 in Figure 12). When the segment has been transferred, IPC compares the computed CRC with the CRC sent in the CCM. If a discrepancy exists in the computed CRC's, the transfer of the segment is marked failed, and the status returned to the sender for retry. If the CRC's match, then a success status is returned. In either event, the status of the transfer is returned back to the sender in a CCM sent from the receiver (9 in Figure 12).

The network transparently passes the status back to the sending IPC facility in a CCM (10 in Figure 12). The sending network processor writes in to the IPC mailbox causing the IPC facility to enter its exception routine again. The CCM is transferred back into the processor (11 in Figure 12) where the `op_code` specifies a returned status. If the status returned is success, the IPC facility checks the work queue for more segments, or messages to send. If no new messages or segments are pending, control is returned back to the application program currently executing.

## 5. Conclusion

In this paper, the functionality and implementation of the real time interprocess communications (IPC) facility developed for the AN/BSY-2 system was described. The IPC facility includes a platform independent message passing interface that supports the unique requirements of a real time distributed system. In addition to timeliness issues, the system requirements also included support for fault tolerance. The IPC facility allows programs to form virtual channels, separating the network specific hardware, and device driver protocols from the application programmer. Group operations are also conveniently defined on logical IDs allowing programs to dynamically register and depart from any group. The low level hardware support designed for supporting fast message transfers and fault detection was also presented. The hardware support included DMA devices, demand driven interrupts and mailboxes, and dual ported memory. The protocols defined for processing incoming and outgoing messages minimized the overhead associated processing the requests. The first system was successfully deployed in 1997.

## 6. Bibliography

- [1] Kang G. Shin, "HARTS: A Distributed Real-Time Architecture", IEEE Computer, pp. 25-35, May 1991
- [2] MPI Users Guide, Draft Proposal 1994
- [3] Saphir, William, "Comparison of Communication Libraries: NX, CMMD, MPI, PVM", Computer Sciences Corporation, NAS User Seminar, November 30, 1993
- [4] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir, "Designing efficient scalable, and protable collective communication libraries", Technical report, IBM T.J. Watson Research Center, October 1992
- [5] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum, "MPI++: Issues and Features", in OON\_SKI '94
- [6] Paul Pierce, "The NX/2 operating system". Proceedings of the third conference on Hypercube Concurrent Computers and Applications, pp. 384-390. ACM Press 1988
- [7] Parasoft Corporation, Pasadena, CA. Express User's Guide, version 3.2.5 edition, 1992
- [8] nCUBE Corporation, NCUBE 2 Programmers Guide r2.0, December 1990
- [9] R. Butler and E. Lusk, Users Guide to the p4 programming system. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992

- [10] Luc Bomans and Rolf Hemple, "The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation in the Intel iPSC/2. Parallel Computing, 15:119-132, 1990
- [11] Gupta, Manish, Banerjee, Prithviraj, "A Methodology for High-Level Synthesis of Communication on Multicomputers", Proceedings fo the ACM International Conference on Supercomputing, Washington D.C., July 1992
- [12] Agarwal, A. Kubiawicz, John, Kranz, David, Lim, Beng-Hong, Yeung, Donald, D'Souza, Godfrey, and Parkin, Mike,"SPARCLE: An Evolutionary Processor Design for Large-Scale Multiprocessors", IEEE Micro, June 1993
- [13] Daly, William et al., "The J-Machine: A Fine-Grain Concurrent Computer", Proceedings of the IFIP 11<sup>th</sup> World Congress, New York, 1989, Elsevier Science Publishing
- [14] "Document for the Real-Time Message Passing Interface (MPI/RT-1.0)", March 6, 2000
- [15] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister, "System Architecture Directions for Networked Sensors", Proceedings of the 9<sup>th</sup> International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX) Vol. 35, Num. 11, November 2000, pp. 93-104
- [16] <http://www.dote.osd.mil/reports/FY97/navy/97ssn21.html>
- [17] <http://www.lockheedmartin.com/files3/lmtoday/9708/seawolf.html>
- [18] <http://www.sei.cmu.edu/publications/documents/92.reports/92.tr.032.html>
- [19] Ada and Beyond, Software Policies for the Department of Defense; NRC, 1997 Copyright National Academy of Sciences, National Academy Press, Washington D.C., 1997 <http://sw-eng.falls-church.va.us/nrc/nrc-contents.html>
- [20] <http://www.naval-technology.com/projects/seawolf/index.html>