Active Message Applications Programming Interface and Communication Subsystem Organization

Alan Mainwaring and David Culler
{alanm, culler}@cs.berkeley.edu
Draft Technical Report
Computer Science Division
University of California at Berkeley

ABSTRACT

High-performance network hardware is advancing, with multi-gigabit link bandwidths and sub-microsecond switch latencies. Network-interface hardware also continues to evolve, although the design space remains large and diverse. One critical abstraction, a simple, portable, and general-purpose communications interface, is required to make effective use of these increasingly high-performance networks and their capable interfaces. Without a new interface, the software overheads of existing ones will dominate communication costs, and many applications may not benefit from the advancements in network hardware.

This document specifies a new active message communications interface for these networks. Its primitives, in essence an instruction set for communications, map efficiently onto underlying network hardware and compose effectively into higher-level protocols and applications. For high-performance implementations, the interface enables direct application-network interface interactions. In the common case, for applications exhibiting locality in communication, these interactions bypass the operating system. To enable the construction of large-scale, general-purpose systems, the interface supports the protected multiprogramming of many users onto finite network resources.This document also describes a prototype system that uses the virtual-memory facilities of the Solaris operating system to implement *virtual networks* that support protected, network multiprogramming. The system caches the active communication endpoints in network-interface memory and demand-pages them between the host and network-interface memories.

Table of Contents

List of Figures

1 Introduction

Switched network hardware is advancing, with increasing link bandwidths and decreasing switch latencies. Network interface hardware is also developing, although the design space remains large and diverse. A growing class of networks, such as Myrinet [9] and TNet [10], combine features of local area networks (LANs) and massively parallel processor (MPPs) interconnects. The new networks take raw performance and scalability from MPPs and add the flexible topologies and administrative properties of LANs, allowing them to be deployed beyond backplanes and machine rooms. These networks fill a technological niche between traditional MPPs and LANs - they tightly connect the components of a single but physically distributed system. Myrinet and TNet, for example, are described as *campus area* and *system area* networks, respectively, characterizations that emphasize their geographic scale and function. Others describe them as *trusted area* networks and emphasize the potential hardware and software optimizations within a single protection domain.

But without low-overhead communication primitives, many applications cannot benefit from these advances in hardware. The increasing bandwidths of network links and the decreasing latencies of network switches cause software overheads to dominate communication costs. This issue has been studied in the context of message passing libraries and their alternatives for MPPs [2] and in the context of TCP/IP performance for LANs [30]. In both of these domains, small messages are common, and software overhead determines their performance. To the extent that small messages are necessary, overhead in communication software determines network utilization and performance. For example, a TCP/IP implementation with 100 microseconds of software overhead per message on a 1Gb/s network must send ~13KB messages for the data transfer time to equal the software overhead. This effect grows more severe with higher link bandwidths. Moreover, the performance of round-trip communication is sensitive to process scheduling: short messages benefit when both applications execute concurrently, and similarly, zero-copy transfers of large messages benefit when the necessary memory resources are scheduled concurrently.

As networks improve, their interfaces, resources, organization, and integration into their host computers becomes increasingly important. Figure 1 shows a common, contemporary network interface organization. The key organizational feature is the capacity of applications to interact directly with the network interface using load and store instructions, thus avoiding the increasingly expensive operating system interactions and keeping the critical, common-case paths simple and fast. The component that enables this is an embedded message processor (or controller) on the network interface. These processors can execute instructions from either on-board or host memory. To varying degrees, these embedded processors operate independently from their hosts, depending upon their computational power, memory capacities, and I/O facilities. Equipping network interfaces with embedded processors or controllers in this fashion permits communication and computation to overlap. The Meiko CS-2 [31], Myricom LANai [9], and Fore ATM SBA-200 [32] network interfaces exhibit this organization.[1] The interfaces operate autonomously, continually attending to the network and processing arriving

1. The Intel Paragon [4] illustrates a variation where the on-board SRAM is removed and a complete i860 microprocessor on the main memory bus is used as a dedicated message processor. The key distinctions are the complete operating system on the message processor and that sending or receiving messages involves one or more main memory bus transactions.

messages. These interfaces can also delegate infrequently used functionality to the host to optimize the common case.

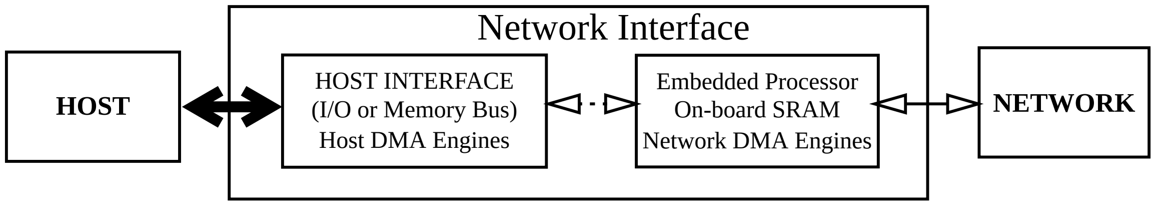

Figure 1 A common contemporary network interface organization

The network interface hardware and firmware together with the host support software realize an *active communication agent*. The network is best understood as a shared, physical resource arbitrated by these agents at its physical endpoints. Within the network, translation of message addresses, authentication, and routing is sought to be done *quickly* - in tens to hundreds of machine cycles. Because the agents form a trusting and cooperative group, they can condition network traffic in a variety of ways. For example, they can prevent traffic from one process from deadlocking traffic of unrelated processes. Agents can also route messages around switch or link failures when necessary, and detect and return undeliverable messages to their senders. In addition, agents can automatically adapt and reconfigure the network as hosts, switches, and links are added or removed from the system; network topology can change dynamically as a result of hardware errors or hardware maintenance. Finally, agents can schedule and multiplex access of applications to the shared network, avoiding deadlock and providing protection between independent processes.

Figure 2 shows one possible organization of a communication subsystem. The quadrants represent the user, the kernel, the network interface, and the network components. The arrows show the relationships among them. The kernel mediates some operations, such as managing the memory mappings for DMA transfers. The physical security of these emerging networks is higher than in conventional ones. Ultimately, however, their physical security is at worst an assumption and at best constructed with hardware and software. Several research projects studied the construction of physically secure networks and distributed systems using secure coprocessors [11, 12]. Their principal conclusion is that the physical security of the network is inseparable from that of its hosts and their operating systems. In their systems, tamper-proof, physically-secure coprocessors enable a variety of techniques for guaranteeing the authenticity of the host operating systems, for authenticating the identity of remote workstations, and authenticating applications, such as remote file servers. The coprocessors store cryptographic checksums and authentication puzzles used in zero-knowledge authentication, key-exchange, and key-agreement protocols. The keys are used to encrypting traffic with private-key encryption.

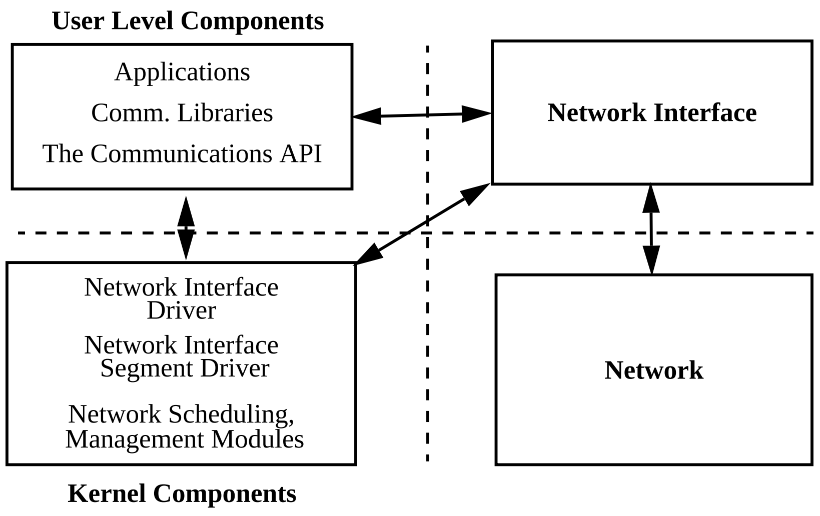

Figure 2 One possible organization of a communication subsystem

1.1 The elusive communication abstraction

One critical abstraction, a simple and general-purpose communications interface, remains elusive. Such an interface should provide primitives that form a portable instruction set for communication. The primitives should map efficiently onto lower-level network hardware *and* compose efficiently into higher-level protocols and applications. The crucial aspect of such interfaces is their integration into an operating system. To promote construction of large-scale, general-purpose systems, they must support protected multiprogramming of a large number of users using finite network resources. The communications API defines the contract between applications and the communications subsystem. A good API not only walks the fine line between portability and generality, but also allows for specialization and optimization by implementations. To do so, the API's primitives must be efficiently implementable on the underlying hardware, with support from the kernel as necessary. The interface should also support a broad spectrum of potentially fault-tolerant or highly-available applications such as file systems, operating systems, client/server programs, peer/peer programs, and parallel programs. And finally, the applications, libraries, and protocols using the primitives must benefit in terms of performance, simplicity, and maintainability. Much of the current network, network interface, and workstation technology is capable of providing at least circa 1992 MPP performance of about 15 microsecond one-way times for small messages and roughly 20-50 MB/s bandwidths for large messages. This performance should track improvements in networks, network interface, and processor technologies.

1.2 A new active message interface

Active messages are a proven and powerful paradigm for constructing high-performance communication protocols, run-time environments, and message passing libraries on massively parallel processors. Active message interfaces of the past were near-misses with respect to supporting the broader application spectrum that are now of interest. Readers unfamiliar with previous active message interfaces and systems can consult a variety of references [1, 2, 3, 4, 5, 6, 7]. Given the increasing presence of high-performance networks, such as Myrinet, TNet, and ATM, the time is right to generalize active message technology and to bring it into the mainstream of networked computing. To achieve this goal requires innovation in four areas: (1) overhauling the active message naming and protection models on which higher-level and system-specific models are constructed, (2) developing communication error and fault models to support fault-tolerant and highly-available appli-

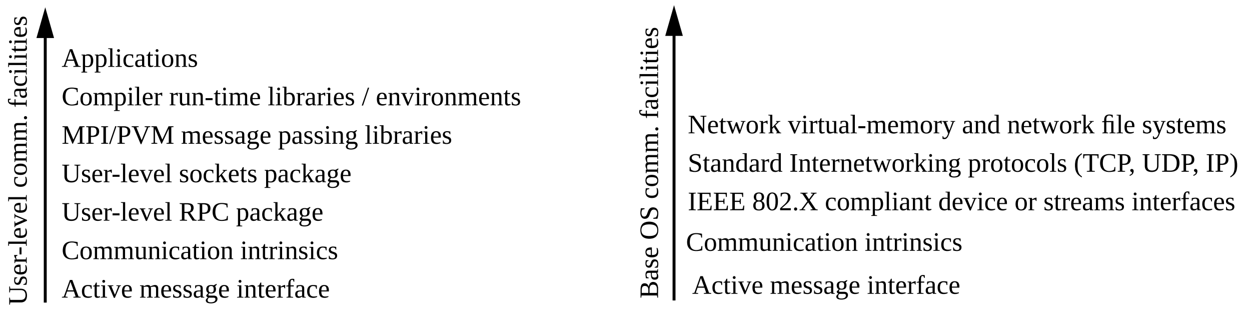
User-level comm. facilities
Applications
Compiler run-time libraries / environments
MPI/PVM message passing libraries
User-level sockets package
User-level RPC package
Communication intrinsics
Active message interface
Base OS comm. facilities
Network virtual-memory and network file systems
Standard Internetworking protocols (TCP, UDP, IP)
IEEE 802.X compliant device or streams interfaces
Communication intrinsics
Active message interface

Figure 3 Building software using active messages

cations, (3) integrating active message operations, communication events, and threads in a simple and cohesive model, and (4) supporting the protected multiprogramming of a large number of users with finite, physical-network resources. The resulting API discussed in Sections 2, 3, and 4 addresses these requirements. The interface is portable, and allows optimizations for systems with high degrees of trust or versatile network interfaces.

A key implementation issue for the new, extended, active message interface is multiplexing a large number of protected endpoint objects onto each physical network interface. Section 5 examines this problem in the context of the Solaris operating system and Myricom network interface. Given this network interface and its 128KB to 512KB of on-board SRAM, we present a prototype system that dynamically *page*s endpoint backing storage between host and interface memory. When *cached* in the interface memory, the memory-mapped endpoint objects allow direct interaction between the network and applications that bypass the operating system in common cases. This keeps latency-sensitive, critical paths simple and fast. The embedded processor handles messages for endpoints in the network-interface memory more efficiently than either it or the host handles messages for those residing in main memory. The host processor assists with handling messages for uncached endpoints. The Solaris segment drivers that manage address spaces provide all the necessary mechanisms for *caching* endpoints in the network interface's SRAM and *demand-paging* them between the network-interface and host memories. Virtual networks lead to several intriguing research questions. For example, what are good endpoint cache-management and endpoint-paging policies in a general-purpose system running a mix of sequential and parallel programs?

To put the active message API in perspective, it is helpful to consider the layers of user and kernel software that use it. At user-level, the primitives should efficiently compose into communication intrinsics, such as barriers or parallel-prefix operations, as well as standard RPC, socket, and message passing libraries. In the kernel, the primitives should support emulation of Ethernet devices, as well as kernel modules, such as distributed virtual memory systems, distributed file systems, and distributed page caches. Figure 3 shows depicts several possible layers of communication software layered on top of the active message interface. Each layer draws upon the services of any lower one, directly or indirectly, as well as the communication services of the base operating system. Note that the user-level communication primitives coexist with and are orthogonal to those of the base operating system.

Figure 3 also shows possible layers of kernel communication software. For example, a network virtual memory system or distributed file system can make use of any communication facilities built using an active message interface, directly or indirectly, as well as the standard kernel communication facilities. A common implementation strategy is to build an IEEE802.3-compliant network-interface driver for the "fast" network. This driver looks like a standard Ethernet device to the higher layers of the TCP/IP protocol stack but uses active message transport operations to transport data. For example, this approach allows the standard protocol stacks to use active messages transparently to communicate between hosts on the fast network. Communication with external hosts uses IP routing and gateway facilities and either traverse the fast network to a gateway or use a local interface connected to other subnets. Another possible implementation strategy for networks with independent, virtual-communication channels is to allocate a percentage of them for active message communications and the remainder for TCP/IP.

1.3 Notational conventions and document organization

The following conventions are used throughout the document:

- Functions are denoted by italicized names followed by parentheses, such as *AM_Poll()*.
- Compile-time constants are typeset in capitalized strings, in a small font, with an AM prefix (AM_CONSTANT). These constants are used and defined throughout the document. A consolidated list of them appears on page 35.

The remainder of the document is organized as follows. Section 2 discusses communication endpoints and bundles and their associated naming and protection model. Section 3 explains the relationships among transport operations, active message handlers, endpoints, bundles, threads, and events. Section 4 defines the semantics of the active message transport, delivery and error handling. Section 5 presents the design of a prototype system for the dynamic management of endpoints, which caches active endpoints in network-interface memory. Section 6 presents related work on network interfaces, communication abstractions, and operating support for high-performance communications. Section 7 presents conclusions, the current status of the prototype system, and plans for future implementations. Section 8 acknowledges the many contributors to the active message specification, and other noteworthy contributors. Appendix A defines the Active Message API, while Appendix B defines the Endpoint and Bundle API.

2 Active Message API: Endpoints and Bundles

This section begins the specification of the new active message API. It generalizes previous active message interfaces to address the requirements and constraints of a broader spectrum of mainstream applications, including client/server programs, operating systems, parallel programs, distributed servers, and parallel clients. Accommodating this diverse application space requires innovations in four areas: (1) naming and protection models, (2) error and fault models, (3) integration of transport operations, communication events, and threads, and (4) supporting protected multiprogramming of a large number of users with finite network resources. Although many issues influence the new interface, it retains the simplicity that made previous active message implementations efficient.

The limitations of previous interfaces and implementations become clear when examined in the context of more general applications. The naming models, for example, in previous active message interfaces were sufficient only for single-program multiple-data (SPMD) parallel programs. This is because all active message communication was constrained within individual parallel processes. Each member of a P-process parallel program had one network port with a unique network address between 0 and P-1. Moreover, instances of a parallel process were mutually trusting and executed inside of a single, externally-managed protection domain. Although simple and elegant, these implementations were incapable of supporting client/server applications, let alone more general communication.

The error and fault models in previous interfaces were sufficient for applications that were satisfied with fail-stop semantics, meaning that any application error that caused the entire process to hang or to abort. The hosts and the interconnect were assumed to be reliable but failures caused the system or system partitions to crash. Though fail-stop semantics are sometimes tolerable for scientific applications, they fail to support fault-tolerant and highly-available applications. Some systems, however, provided application checkpoint and restart, but none of the active message implementations were compatible with them.

Previous active message interfaces supported only single-threaded applications, but determined programmers with sufficient knowledge of implementation details used active messages in multi-threaded applications in ad hoc manners. Another limitation of previous systems was that asynchronous, communication-event handling was unsupported or was supported using nonstandard interrupt facilities. Processes were required to poll for messages on systems where message interrupts or signals were unavailable. Some systems provided specialized, i.e., nonstandard, message interrupt facilities, which were frequently difficult or expensive to use. For example, some interfaces imposed arbitrary time limits on the execution of active message handlers or restricted handler computations.

The new interface, fortunately, is a superset of its predecessors. That is, it permits sequential and parallel processes to create multiple communication endpoints. Endpoints bear some resemblance to conventional sockets or ports, but there are important differences. Any endpoint can name and send messages to any endpoint subject to a protection mechanism using endpoint tags. Endpoint tags and endpoint names form capabilities that control communication among endpoints. Although hosts and the interconnect are nearly reliable, when faults or errors do occur some active messages can become undeliverable and are returned to their senders. This allows application-specific recovery procedures to deal with returned messages. In addition, the new interface supports multi-threaded applications. Asynchronous events use thread-synchronization variables and event masks that select which communication events post the synchronization variables.

But broadening the appeal of active messages to mainstream applications and continuing high-performance implementations are often conflicting goals. To minimize this conflict, the new active message interface was formulated with an eye towards how its organization and functionality impact high-performance implementations. Although the restrictions and limitations of previous interfaces made their implementations simple and efficient, the same restrictions and limitations prevent them from supporting the broader spectrum of applications now required. The new interface, however, makes the necessary and sufficient generalizations in order to satisfy the requirements of the new applications. Future revisions to the new interface will address changes derived from experiences with initial implementations; then, features initially postponed and other functionality will be considered. (Section 7.2 on page 31 discusses features and communication models currently postponed and omitted from the interface.)

2.1 Communication endpoints

The active message interface allows applications to communicate among objects called *communication endpoints*. Each endpoint contains all state associated with an independent, user or kernel network "port." The aggregation of these related resources into a single object enables simple naming, protection, and management. Furthermore, any two endpoints can communicate, although this is contingent upon protection and authentication checks. This means, among other things, that active message communication can now cross traditional protection boundaries and domains; it is no longer restricted to use among the members of individual, user-level parallel processes. Two examples show the new interface's flexibility: a kernel, file-system endpoint can communicate with a user-level, client endpoint, and an endpoint in a sequential process can communicate with an endpoint in a parallel process.

Endpoint components are easily understood by their use in active message functions, which are two-phase, request and reply operations between pairs of endpoints. As shown in Figure 5, an endpoint has a *send pool*, a *receive pool*, a *handler table*, a *virtual-memory segment*, a *translation tabl*e, and a *tag*. An active message is sent from an endpoint send pool to an endpoint receive pool. It carries an index into a handler table that selects the handler function for the message. Upon receipt of a request, a request-handler function is invoked with a small number of arguments; likewise, when a reply is received, the reply-handler function is invoked with a small number of arguments. The bulk data transfer functions copy memory from a sender's virtual address space to the receiving endpoint's receive pool or virtual-memory segment and deliver an associated active message when the transfer is complete. Although endpoints have globally unique names, it is convenient for applications to have compact, local names for remote endpoints. An endpoint's translation table associates small, integer indices with the names of destination endpoints and their tags. A translation-table index is a compact, relative name for a destination endpoint and its tag. Each tag is a 64-bit integer. An endpoint can send an active message to another endpoint if and only if the tag in the sender's translation-table entry for the destination endpoint matches the destination endpoint's tag at the time the message is delivered or if the destination endpoint's tag is a wild card (a special value that matches all tags).

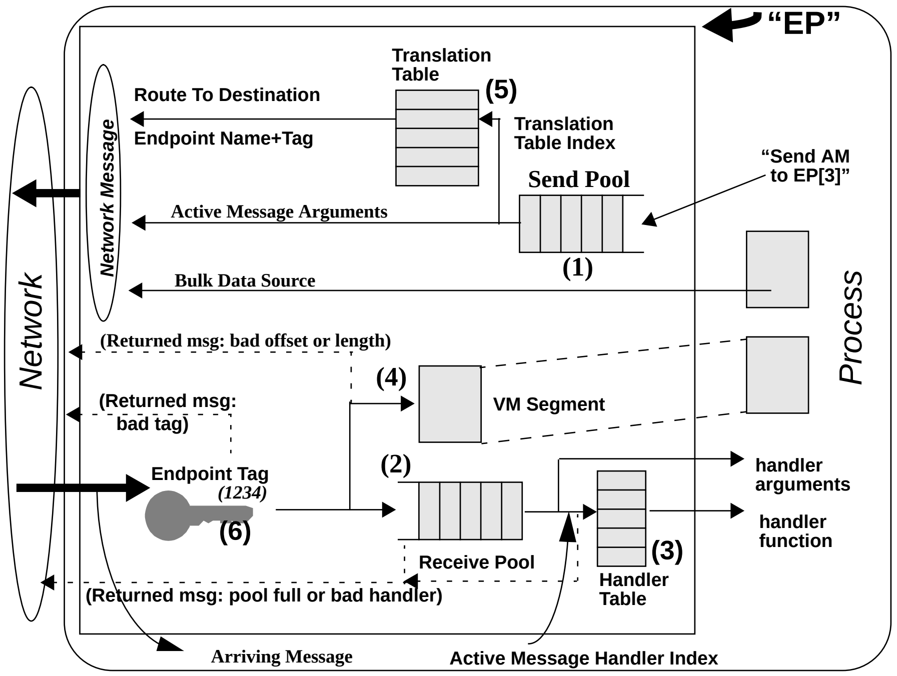

Figure 4 Anatomy of an endpoint

2.2 Components of communication endpoints

Each communication endpoint has the following components, whose numbers below correspond to the ones in Figure 5.

Component #1:A send pool for sending messages from the endpoint.

This is the buffer and control state for sending messages. Send pools are not directly exposed to applications. Because libraries or a communications system may randomly access send-pool entries, send pools are "pools" instead of FIFOs.

Component #2:A receive pool for receiving messages into the endpoint.

This is the buffer and control state for receiving messages. Receive pools are not directly exposed to applications. Because libraries or a system may randomly access receive-pool entries, receive pool are "pools" instead of FIFOs.

Component #3:A handler table for translating handler indices into functions.

Active messages carry indices into handler tables, a form of indirection that affords a level of protection between senders and receivers, and eliminates the requirement that senders know *a priori* addresses of handlers in other processes.

Component #4:A virtual-memory segment that receives memory transfers.

Each endpoint has one application-specified, virtual-memory segment for receiving bulk transfers. Its base address and byte length specify a window into the receiver's address space into which endpoints with valid tags can write.

Component #5: A translation table that associates indices with global-endpoint names and tags.

The translation table is an array that associates indices with global-endpoint names and their tags. Operations on table entries are atomic with respect to their use.

Component #6: A tag for authenticating messages arriving at the endpoint.

Tags are 64-bit integers that authenticate messages arriving at an endpoint. Applications manage tags and may change them at any time. All implementations define special tags to accept all messages (AM_ALL) and to accept no messages (AM_NONE).

All endpoint components have default sizes and values when created. The handler and translation tables start with 256 entries, but applications can dynamically resize either table as necessary. All handler-table entries point to the *abort()* function; all translation-table entries are marked as unused. The virtual-memory segment base address is set to zero, and its length is zero bytes. The endpoint tag is set to AM_NONE in order to prevent messages from being delivered until the endpoint is readied for use.

Preparing an endpoint for use requires initializing handler-table entries, setting the endpoint tag, establishing translation-table mappings to destination endpoints, and (optionally) setting the virtual-memory segment base address and length. Applications can dynamically resize the send and receive pools to reflect their anticipated resource requirements, although the system accommodates such requests only when resources are available. When they are not, it may return messages due to persistent congestion at particular endpoints. Section 4.4 on page 21 provides complete details about allocating adequate endpoint resources in order to prevent messages from being returned due to persistent congestion.

2.3 Communication endpoint bundles

An endpoint bundle is a set of endpoints created by one process. These endpoints are treated as a single unit for communication, event management, and synchronization. Any process can create multiple endpoints and gather related ones into endpoint *bundles*. UNIX processes, for instance, can create multiple sockets and perform operations on aggregates of them. Similarly, Mach tasks can create multiple ports and organize them into port sets that share message queues. Active message programs can also create multiple endpoints, gather them into endpoint bundles and succinctly perform operations on the bundles. The motivation for creating multiple endpoints is to avoid undesirable interactions that arise when independent software packages use a single shared port. The motivation for bundles is to support the need of single-threaded programs for operations on aggregates of endpoints. Section 3.3 shows that endpoint bundles address deadlock issues that arise when multiple, independent software packages are composed in a single-threaded application.

2.4 Components of communication endpoint bundles

An endpoint bundle has the following components, whose numbers below correspond to the ones in Figure 5 on page 9.

Component #1: A collection of endpoints.

At any given time, an endpoint is a member of exactly one bundle. Applications can move endpoints between bundles.

Component #2: A *thread-synchronization variable (a binary semaphore).*

When any endpoint in a bundle generates an event, the thread-synchronization variable is posted. The identity of the specific endpoint in a bundle responsible for generating an event is unavailable to the application.

Component #3: An *event mask*, which is under the control of the application.

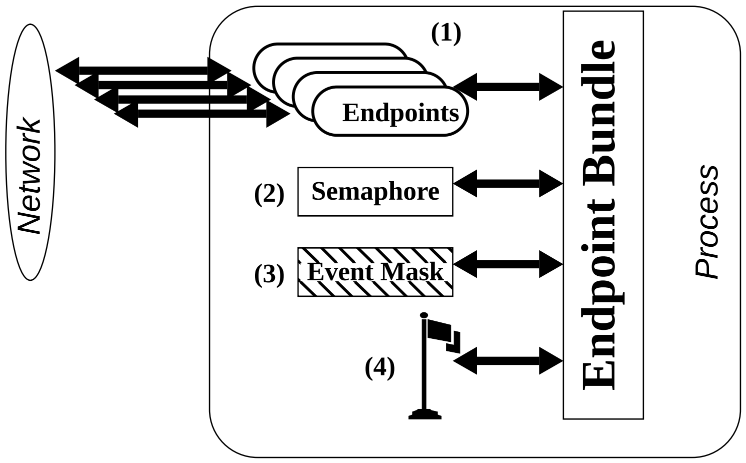

Figure 5 Anatomy of a bundle

The event mask selects which endpoint states or state transitions generate events and post the synchronization variable. For example, the mask enables empty receive pools that becomes nonempty to generate an event.

Component #4: An *access mode flag* that indicates if concurrent use of the bundle or its endpoints is expected.

The flag informs a system if multiple, application threads will access the bundle or its components concurrently. By default the system automatically serialize all accesses, but applications that promise serial access can override the default.

All bundle components have default values at the time they are created. The event mask is set to disable the generation of all events. The access mode flag is set to serialize concurrent accesses to a bundle and its components (access mode is AM_PAR). Applications can set the access mode flag to indicate that only sequential accesses to the bundle and its components will occur (access mode is AM_SEQ); the system then performs no automatic serialization or locking. Implementations can optimize performance for sequential access, or they can optimize performance by unconditionally serializing all accesses (and ignoring the access mode flag altogether).

Transport functions that send active messages, for example *AM_Request4(),* can receive and handle messages arriving at other endpoints in the bundle containing the endpoint sending the message. Similarly, transport functions that receive active messages, for example *AM_Poll()*, receive and handle messages arriving at any endpoints in a specified bundle. Section 3.2 on page 15 discusses how these semantics facilitate the composition of software packages in single-threaded programs.

Each bundle has a unique event mask and synchronization variable, which is a *binary semaphore*. The bundle is the scope of the synchronization variable and the event mask. The event mask is a bit vector that select which endpoint-state transitions generate events and post the synchronization variable. The endpoints in a bundle share a common event mask and synchronization variable; the identity of the endpoint producing an event is unavailable.

2.5 Communication endpoint naming

Several issues and goals influence the endpoint naming model, the most important of which is to maintain independence between the interface and implementation-specific representations of endpoint names. This separation allows multiple, higher-level naming systems to be constructed on top of the primitive one in the interface. The second most important issue is that it is convenient to assume that endpoints have globally unique names, while retaining their ability to locally refer to endpoints with small integer values regardless of whatever higher-level naming system is used. The third most important issue is that position-independent endpoint names facilitate building systems that migrate endpoints.

All endpoints have globally unique names within a system, but these names are not necessarily in a form that is convenient or easily manipulated. Rather than defining the specific representation of a global-endpoint name, the interface treats global-endpoint names as an opaque type and manipulates either the named endpoint objects or indices to the named endpoint objects. The endpoint-translation table accomplishes this through a level of indirection. The interface specification then remains independent of external name servers and global operating systems, although implementations require that external agent(s) exist to manage mappings of global-endpoint names to physical resources.

Applications use the translation table to map small, integer indices to arbitrary endpoint names and tags in the system. This approach preserves the simple naming mechanism of previous active message interfaces in which a set of nodes were named with small integers from 0 to P-1. A process then maps a destination endpoint, given its globally-unique name, by specifying the address of a local-endpoint object and an index into its translation table. Applications can dynamically add and remove translation table mappings.

This model allows for the existence of multiple name managers and naming models. For example, it supports group models where endpoints are members of communication groups and members are named with a group name and a member number. Whereas in connection models, endpoints instantiate the ends of a logical communication channel and channel identifiers name the channel's endpoints. The best naming model is application- and protocol-specific. For example, using groups [13, 14] to name an aggregate of N endpoints can be more convenient and efficient than using N^2 connection identifiers. Because of these sorts of issues, the interface provides a primitive naming mechanism on top of which more specialized naming models can be constructed.

The management of each endpoint-translation table is local. However, multiple applications can coordinate and manage their translation tables to produce familiar environments. For example, a SPMD parallel library or the system function that initiates the parallel program can arrange that index *i* from every endpoint uniformly names *virtual processor i*. This recreates the name space of previous active message systems. But unlike previous systems where processes had one endpoint, this interface allows the number of endpoints in the parallel library or software package to grow and to shrink. Adding endpoints and maintaining a uniform *virtual processor* ranking is straightforward. Removing one or more endpoints, however, can introduce gaps, which could initiate a re-ranking of the remaining endpoints into a new, contiguous-index range.

A position-independent endpoint is one that can be migrated between hosts, either with or without the process that created it. Position-independence affects implementations that support process checkpoint and restart. Although the translation table provides a level of name translation and indirection, the ability to migrate endpoints without leaving residual dependencies and the ability to checkpoint and restart processes depend on the endpoint name server and how it represents global-endpoint names. For example, a name server might represent endpoint names as the triple {IP address, UNIX pid, Endpoint Number}. With the support of a global operating system layer, it might represent them as {GLOBAL pid, Endpoint Number}. The former representation supports endpoint migration less readily because IP addresses are bound to specific network interfaces of hosts [15]. The latter supports endpoint migration more readily because the endpoint names are not bound to specific physical resources. Even with position-dependent names, however, the API as a whole can support process migration and checkpoint/restart.

2.6 Communication endpoint protection

It is desirable to have a simple protection mechanism that controls which messages can be delivered to endpoints and that captures general relationships among endpoints. The mechanism should also maintain independence between the interface and specific models that govern and enforce relationships among endpoints (*e.g*, groups and connections). Given the assumed or enforced physical security of the network and its hosts, the protection model addresses programs that inadvertently send messages to the wrong endpoints and introduces a difficult but not insurmountable hurdle for malicious users who wish to cause havoc.

As with endpoint naming, rather than taking a specific position on the allowable relationships among endpoints, such as legislating a group-based or a connection-based model, the interface abstracts such relationships into tags. These tags implicitly identify sets of communicating endpoints and associate them with effectively unique integers. Tags chosen intelligently from the 64-bit tag space (*e.g.,* such as randomly chosen tags) are very likely unique. Applications can use tags to identify aggregates of endpoints unambiguously. Therefore, tags complement the endpoint naming mechanism in two ways: they identify aggregates of endpoints and provide a simple message authentication model. This approach differs from interfaces such as MPI [16, 17, 18, 19, 20] where data structures represent groups and contexts.

Each entry in a translation table associates its integer index with a global-endpoint name and a tag. An endpoint can then send an active message to another endpoint if the tag in the sender's translation table logically matches the receiving endpoint's tag at the time the message is delivered. There are two special values for tags: a *never-match* one, which, as the name suggests, never matches any tag, and a *wild card* one, which matches all tags except the never-match one. Effective protection from errant messages relies on the use of a sufficiently large and sparse tag space. This makes it highly unlikely that such messages have a matching tag or that correct tag values can be guessed. Other systems use similar authentication mechanisms [21, 22].

At any time, the process that created an endpoint may change its tag. For example, a process may want to prevent messages from a suspect peer from being delivered. The ability to change endpoint tags on demand allows processes to revoke the capability of other endpoints to send them messages. From a receiver's point of view, the change is atomic with respect to communications: once the change is made, the system will deliver messages sent only from endpoints whose translation-table entries have the new tag; messages already in an endpoint's receive pool associated with the previous tag are not returned to their senders. Endpoints with stale translation-table entries must obtain the new tag and update their own translation-table entries before the system will deliver more messages. Messages using stale translations are returned to their senders. Because senders specify tags when creating translations, two endpoints that map a common, third endpoint cannot surreptitiously obtain knowledge of each other's tag. Refer to Section 4.3 on page 19 for details.

Parallel libraries can coordinate the management and distribution of their tags. All endpoints in a software package, for example, an independently-written software library, can use the same tag. Each package can then limit the scope of its communication to its own endpoints. This ability generalizes to multiple, independent software packages composed in one application. Client-server protocols can begin by having clients communicate with a server's well-known endpoint, that the server marks with the wild-card tag. The server can then redirect clients to new endpoints with new tags to allow authenticated communications to proceed. Barring independent application-level or implementation-specific protection measures, the same degree of protection with respect to active message communication (between two endpoints) exists between two libraries within an application as exists between two libraries in different applications.

As mentioned previously, because each endpoint has a private translation table, a collection of clients bound to an *unprotected* but well-known server endpoint with a wild card tag cannot obtain the tags of the other clients. However, because any client can map to any endpoint with a wild card tag, multiple clients can simultaneously interleave writes to a server's endpoint memory segment and corrupt its contents. To cope with this, a server can copy data from endpoint virtual-memory segments into private memory and inspect the contents before proceeding. Corruption is not an issue for the medium and short (*i.e.*, non-bulk transfer) messages because the system passes them by value and atomically adds and removes them from send and receive pools. Still, a suspicious server can implement active message handler functions that first carefully inspect their client-provided arguments before incorporating them into its computation. And, after initial contact, servers can always redirect clients to new endpoints protected with new tags for subsequent communications.

2.7 Managing endpoints names and tags

The new interface specification intentionally avoids any position on how an application obtains the names and tags of remote endpoints. The means by which processes obtain them are intentionally left outside the scope of this specification, as are the interfaces to the endpoint name server(s) and manager(s). By specifying the interface independently of specific endpoint naming representations and specific policies for obtaining endpoint names and tags, we create a more general specification and enable a variety of name-space-management strategies. Many standard procedures and conventions for rendezvous apply. For example, the endpoint name space can be embedded in a shared file system. Endpoint names can also be exchanged between applications using previously agreed upon files, or using conventions such well-known endpoint names.

The specification of endpoint names and tags enables interfaces to be implemented flexibly. This is because the specification permits send-side or receive-side handler argument validation and message authentication. For send-side validation, implementations can check remote endpoint names before sending a message, and for receive-side validation, they can route messages to the appropriate destination network interface, which checks its destination endpoint name before delivering the message. The specification of tags permits implementations to validate tags at either the sender or receiver. For example, implementations can cache remote endpoint tags in local-endpoint translation-table entries and place the burden of checking tags on the senders, or can pass tags in messages and check them at the destination before delivery. This flexibility also allows hybrid strategies in which messages contain names and/or tags until some threshold amount of communication occurs, at which time the sending endpoints cache name and tag validation results. Send-side protection and caching information in sending endpoints can optimize performance for receiver-limited systems at the cost of introducing consistency issues in the presence of communication faults and exceptions. Receive-side protection models are implemented more easily, but can generate longer messages (containing tags and other information) and burden the receiver with more work. The specification allows implementations to consider these trade-offs and to make the best choices.

3 Active Message API: Concurrency and Synchronization

This section discusses the concurrency and synchronization issues that arise when endpoints, bundles, threads, and events are unified. Four scenarios illustrate the semantics of and interrelationships among active message transport operations, active message handlers, and events handling. The basic interactions between threads and endpoints cause active messages to be sent, active message handlers to be invoked, and synchronization events to occur. Understanding these interactions makes the effective use of active messages possible so that deadlock-free applications, libraries, and protocols can be constructed. The four scenarios illustrate specific points, such as when active message handlers execute atomically and when synchronization is required to maintain the consistency of data shared between threads and handlers.

The interface supports both single-threaded and multi-threaded applications. Its specification enables implementations to exploit the performance of multi-threaded and multiprocessor environments, yet it avoids penalizing applications with unnecessary locking or synchronization costs. The active message interface does not specify the interface to the threads package.[2] For single-threaded applications, the interface requires support for blocking the sole thread of computation on a binary semaphore. For multi-threaded applications, the interface requires support for basic thread operations and scheduling, as well as for blocking on binary semaphores. The facilities in the POSIX P1003.4A/D8 subset of Solaris [23, 24] are adequate.

The communication subsystem notifies applications of communication events by using the binary semaphores[3] in endpoint bundles. This integration enables event-driven application execution using standard mechanisms and programming models, by allowing threads to yield their CPUs and wait for communication events rather than continuously polling. Allowing threads to block is essential to support event-driven applications, such as servers. Initially, in order to support event-driven applications, the interface allows threads to yield their CPUs and block while all endpoint receive pools in a bundle are empty. The endpoint bundle event mask controls which endpoint-state transitions generate events. Then, when a state transition occurs that generates an event, the system atomically posts the bundle's binary semaphore and clears the corresponding event mask bit. If there are threads blocked on the semaphore, then one is unblocked.

The endpoint naming and protection models facilitate the development of modular software where independently written software packages remain correct when composed in applications. Each software package tacitly defines a domain of communication among its endpoints. Intuitively, a correctly written and deadlock-free communications package, such as a math library or a byte-stream protocol, should remain deadlock free when executed concurrently in an application with other such packages, providing that all threads make forward progress and consume arriving messages. Writing composeable software with active messages requires that programmers understand how the operations use endpoints and bundles.

Most active message operations are synchronous, that is, they block until the source storage can safely be reused. In order to avoid deadlock, a thread must not only receive requests and replies while sending requests, but it must also receive replies while sending replies. These semantics prevent threads from deadlocking in common situations, for example, when two threads block while exchanging requests and replies between a pair of endpoints. These semantics generalize to endpoint bundles. To avoid deadlock a thread must (1) receive requests and replies on any endpoint in the bundle from which it is sending a request, and (2) receive replies on any endpoint in the bundle from which it is sending a reply. Although the implementations of the request and reply functions take care of this, programmers must be cognizant of this behavior.

The following subsections present scenarios that not only illustrate fundamental issues relating transport operations, active message handlers, endpoints, bundles, threads, and events, but they should also serve as templates for understating other scenarios.

2. The POSIX Threads P1004.4A passed the International Standards Organization CD balloting and become P1004.1C. Draft 10 was submitted in June to the IEEE Standards Boards for approval in June. With standardization of threads packages coming to fruition, perhaps binding this active message interface to this standard is a reasonable way to establish a frame of reference regarding threads. Without such a binding the relationship between the interface and the threads system is vague.

3. The management, methods of presentation, and taxonomy of all the endpoint-state transitions that should generate events are open research issues. For example, only receive pool state transitions currently generate events. Other endpoint-state transitions are interesting, such as send pool transitions from full to not-full or not-empty to empty. It is worth considering generating events when messages arrive with bad tags, offsets outside of virtual-memory regions, or invalid active message handler-table indices.

3.1 Scenario 1: one thread and one endpoint

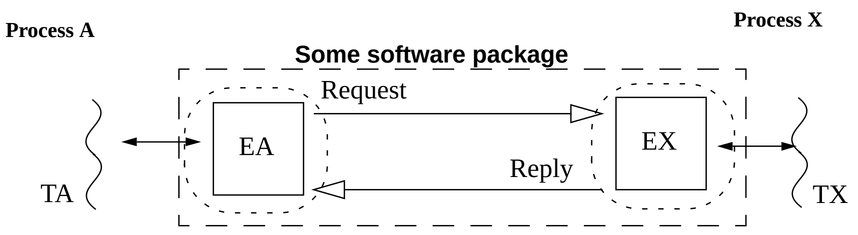

Figure 6 A single-threaded program with a single endpoint

Rectangles with dashed outlines denote software packages, ovals with dotted outlines represent endpoint bundles, and squares are communication endpoints.

3.1.1 Explanation

This scenario formalizes previous active message systems [1, 2] in which each process has a single, logical network endpoint. These previous active message systems supported single-threaded applications only (or multi-threaded applications where only one thread used active messages in an ad hoc manner). In part, this is because these systems either lacked support for event-driven application execution [1, 3] or supported primitive but nonstandard interrupting message facilities [2, 6]. In these systems, events were also de-coupled from thread and process scheduling systems. Experience showed that message polling mechanisms alone inadequately supported server-like applications and computational-intensive code. Moreover, inserting polling points complicated algorithms and degraded performance. Alternatives such as customized, user-level interrupts made existing programming models more complicated.

3.1.2 Implications

These factors made the new interface explicitly support single and multi-threaded applications, abstracting the single, physical network port of previous systems into more flexible, multiple, communication endpoint objects, and integrating endpoint events with standard, thread-synchronization mechanisms. This in turn enables threads to block on the binary semaphore that is posted when events occur under the control of an *event masks.* (The bundle is the scope of the semaphore and event mask.) Any unmasked event occurring on any endpoint in the bundle posts the semaphore. A single thread blocked on the semaphore is unblocked and can receive the waiting message by polling (see *AM_poll() in* Section A.12). To avoid deadlock in this scenario, threads TA and TX must receive requests and replies while simultaneously exchanging messages.

3.1.3 Handler semantics

In this example, as in previous active message systems, handlers execute atomically with respect to other handlers. Because of the single-threaded environment, handler functions can avoid explicitly serializing accesses to objects shared with other handlers or with the main thread of computation. This is because the thread manages critical sections simply by not calling any communication functions within the bounds of the critical section (*i.e.,* by not polling or sending messages). Consequently, handlers remain easy to write, and it remains easy to share counters and data structures between handlers and the main thread of computation. A few restrictions on handlers remain in this case: handlers cannot block, request handlers must reply exactly once to the requesting endpoint, and reply handlers cannot poll, send, or receive messages.

3.2 Scenario 2: independent instances of scenario 1

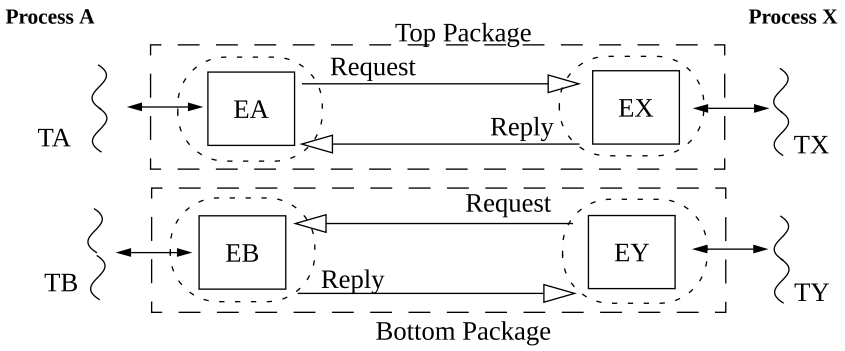

Figure 7 Multi-threaded program with multiple, independent endpoints

Rectangles with dashed outlines denote software packages, ovals with dotted outlines represent endpoint bundles, and squares are communication endpoints.

3.2.1 Explanation

This example illustrates two independent and concurrent instances of the first scenario: two processes, each composed of two, independent software packages, each independently written, and each with one thread per endpoint. Providing that all threads are eventually scheduled and each thread services its associated endpoint then communication remains deadlock-free. Within each process, however, other dependencies among threads can exist and produce deadlock independent of communication structures. But such issues always exist and are outside the scope of the interface.

3.2.2 Implications

This example shows the need for multiple, distinct, synchronization variables in order to direct events to the appropriate threads. In process A, threads TA and TB can restrict their event handling to their respective bundles, largely because judicious use of endpoint tags allows packages to contain their messages among their endpoints. For example, endpoints in a package could have the same tag or a distribution of tags known only within the package. Each endpoint has a distinct handler table and virtual-memory segment managed independently. Hence, the collection of endpoints contained in a package define a “context” for performing communication independently from other packages.

3.2.3 Handler semantics

Because this example consists of multiple, independent instances of the first scenario, similar restrictions apply to handler functions. For instance, providing that the same handler is not registered in multiple endpoints, active message handlers remain atomic relative to the other handlers in each bundle as well as the thread associated with each bundle. Again, providing that the packages are independent, threads can perform communication operations and execute handlers concurrently without synchronization.

If either the threads or the active message handlers within a process share data structures, then steps should be taken to synchronize accesses to those structures. Note that it is permissible for request handlers associated with one bundle to call functions that send requests using endpoints in some other bundle, so long as cyclic dependencies among all bundles involved are avoided. This prevents nesting of handlers and nonatomic, handler invocation with respect to the handlers of any particular endpoint. If request handlers follow this restriction and nonatomic, handler function execution is anticipated, then the system can deliver all messages and remain deadlock free.

3.3 Scenario 3: one thread with multiple endpoints

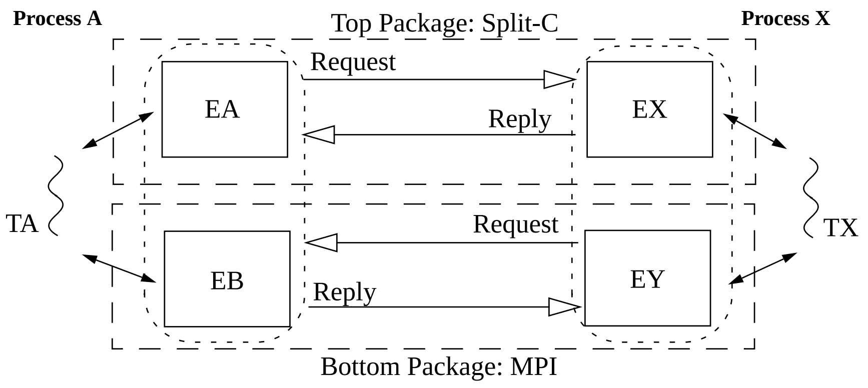

Figure 8 A single-threaded program with multiple endpoints

Rectangles with dashed outlines denote software packages, ovals with dotted outlines represent endpoint bundles, and squares are communication endpoints.

3.3.1 Explanation

This case shows a single-threaded application with multiple, independent software packages - instances of the MPI and Split-C libraries. If thread TA blocks sending a Split-C request from EA to EX, and thread TX blocks sending an MPI request from EY to EB, then deadlock results unless TA handles requests at EB and TX handles requests at EX. In general, to prevent deadlock when independent software modules are composed in a single-threaded process, active message operations must handle messages in *endpoint bundles*. This is the original motivation for the abstraction of endpoint bundles.

3.3.2 Implications

Unlike the example in Section 3.3.1, where there is a one-to-one correspondence between threads and endpoints, in this case thread TX handles events from both endpoints EX and EY. Hence, bundle synchronization variables cannot be associated one-to-one with endpoints. This scenario also shows that multiple software packages can be composed in a single-threaded application and retain much of their independence. This is because each package has its own endpoints and each endpoint has a distinct handler table, virtual-memory segment, and tag. A package can then use tags to contain its messages within its endpoints and to prevent its messages from straying into other endpoints and packages.

Although the two packages *should* be written independently and without knowledge of each other, composing them in a single-threaded application creates the possibility of cyclic communication dependencies. To break such cycles, active message operations on an endpoint handle messages on all endpoints in the same bundle. In this way, multiple packages can add their endpoints to a common bundle and know that their messages will be handled.

3.3.3 Handler semantics

The same restrictions on handlers from scenario #1 apply in this scenario as well.

3.4 Scenario 4: multiple threads with one endpoint

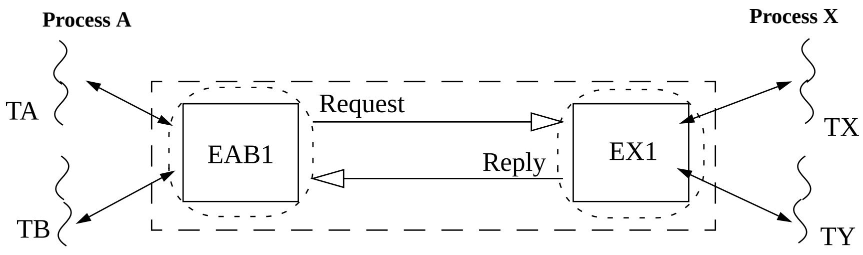

Figure 9 Multi-threaded program with one, shared endpoint

Rectangles with dashed outlines denote software packages, ovals with dotted outlines represent endpoint bundles, and squares are communication endpoints.

3.4.1 Explanation

This example shows multiple threads in each process sharing a single endpoint in one package. Within each process, there are two basic strategies. Threads can synchronize their operations upon the shared endpoint and bundle using mutex locks of their own creation and management. If this is done, the rules and regulations governing handlers from scenario #1 continue to apply. Otherwise, multiple threads can perform concurrent operations upon the endpoint and bundle. The system guarantees that messages are atomically added and removed from send and receive pools and that handlers are executed with correct arguments. However, handlers may not be executed atomically with respect to other handlers.

3.4.2 Handler semantics

If multiple threads operate concurrently on a shared endpoint, then the endpoint's handlers may no longer execute atomically with respect to the endpoint's other handlers. To avoid confusion *in this case*, applications should synchronize accesses to objects that are shared among handlers and threads. While handlers are no longer atomic and non-preemptable, multiple threads can handle incoming active messages in parallel with increased bandwidth. For example, multiple-client processes could send requests to a single, multi-threaded server. The number of threads in the server can be dynamically adjusted to accommodate different client-communication request rates and loads. If each client request can be handled independently of the others, then the functions can execute concurrently. If, on the other hand, handling each request requires state shared among the other threads or the handlers involved, then serializing those accesses may be required.

4 Active Message API: Transport Operations

The interface provides point-to-point, active message operations between pairs of endpoints that are cast as matching request-and-reply message pairs. The *requesting endpoint* is the one that sends the request, and the *replying endpoint* is the one that sends the corresponding reply. Upon receipt of a request message, its request handler is invoked; likewise, when a reply message is received, its reply handler is invoked. Request handlers must reply exactly once to the requesting endpoint; it is an error to do otherwise.[4] Some implementations, however, will signal this error, but it may go undetected in others. In addition, reply handlers cannot perform

4. The phrases "it is an error to X" and "an error will be signaled if X" have distinct meanings. The former implies that applications written correctly should not do X and that correct interface implementations may report the errors optionally. The latter means that applications written correctly should not do X and that a correct interface implementation must always check for and report such errors.

request or reply operations. Handlers are not explicitly typed as request or reply handlers, but the semantics of the two generally make it convenient to consider them as being so typed.

The interface distinguishes “small” transfers from “medium” and “large” memory-to-memory bulk transfers. For small messages, typically ones contained and composed entirely within a processor’s register set, the interface provides request and reply operations that pass a small number of words (4 or 8, as a result of common function-calling conventions) by value. For medium messages, the interface operations support sending more than a small number (*up to 128*) of words by value. For large transfers, the operations copy source-memory regions to user-specified destination endpoint memory segments and then deliver associated active messages. The active messages are logically delivered *after* the bulk transfers finish writing the destination memory. Handling the active message is the *notification event* associated with a bulk transfer.

Multiple threads can concurrently send requests and replies from an endpoint in a *shared* bundle. This allows multiple threads to poll concurrently a shared bundle and to handle individual incoming messages in parallel. The system performs the necessary synchronization to serialize accesses to endpoint and bundle components, such as send and receive pools, while attempting to maximize potential concurrency. Because sending requests or replies from an endpoint can cause incoming messages on any endpoint in the same bundle to be handled, concurrent sends from an endpoint can cause concurrent message handling. Concurrent operations can also cause nonatomic handler execution.

The system passes all active message request- and reply-handler functions a *token*. A token is an opaque pointer to data that identifies the source and destination endpoints. *AM_GetSourceEndpoint()* translates a token into the globally-unique endpoint name of the sending endpoint. *AM_GetDestEndpoint()* translates a token into the globally-unique endpoint name of the receiving endpoint. *AM_GetMsgTag()* retrieves the message tag from a token.

4.1 Active message semantics

The simplest model for correct handler execution is the following. Reply handlers should not call any functions directly or indirectly that poll, send requests, or send replies. Request handlers must reply exactly once to the requesting endpoint; it is an error to do otherwise. Additionally, functions that send active messages, like *AM_request_4()*, can receive and handle active messages arriving at any endpoint in the same bundle. For example, given a bundle with endpoints A, B, and C, sending a request message from A can handle arriving request and reply messages to A, B, or C. Sending a reply message from A can handle arriving reply messages to A, B, or C. These rules produces correct handler behavior and allow for the modelling of request and reply handler nesting. While sending a request, a request message can be received and handled. The request handler must then issue a reply, which can in turn receive a reply message and invoke a reply handler. Thus, in this scheme, the maximum nesting of request, reply and handler functions on a thread’s call-stack is four deep.

Advanced active message applications can make use of the following semantics. In addition to the single mandatory reply, request handlers *can* send request messages using bundles that are distinct from the requesting and replying bundles. If request handlers restrict such use to acyclic chains of bundles, and if nonatomic handler execution is anticipated, then the system can deliver all messages and remain deadlock free. The effective use of this semantics requires that programmers have high-level knowledge of the dependencies among the relevant endpoints, bundles, and software packages. Although this semantics is more complicated than that stated in the previous paragraph, it is nevertheless valuable. For example, it permits package A to export a *printf* function built on top of request and reply operations that can be called from request and reply handlers in package B, providing that the endpoints and bundles in packages A and B are distinct. The maximum nesting depth of request and reply operations and handlers is easily modelled in this case: it is twice the length of the chain of requests from endpoint to endpoint in distinct bundles, plus two. Note that when the “deepest” request handler replies, it can receive and handle a reply message.

4.2 Bulk request and reply operations

The bulk transfer request and reply functions require additional arguments. These arguments specify the source-memory address, the destination-memory offset from the base of the remote reply endpoint's virtual-memory segment (omitted for medium-sized transfers), and the number of bytes to copy. The functions *AM_MaxMedium()* and *AM_MaxLong()* return the maximum transfer sizes for medium and bulk transfers, respectively. The source- and destination-memory segments can have arbitrary memory alignments, although implementations can optimize for page, cache-line, and double-word alignments. Such optimizations can be performed at the expense of word, short, and unaligned transfer performance. From the moment a thread issues a bulk request or reply operation until the moment the receiver invokes its handler, the contents of the destination-memory region is undefined.

4.3 Message delivery and error model

For request and reply operations, a message is considered sent when the source storage (registers or memory) can be reused. A message is considered received when its handler function is invoked in the receiving process. A message is considered delivered once it is entirely written into its destination endpoint's receive pool (which logically precedes its reception and handling). Once delivered, the communication system cannot control when an active message's handler executes. Sending active messages from any endpoint in a bundle can, as a side-effect, receive incoming active messages on all endpoints in the same bundle and execute their handlers.

The system delivers all messages exactly once barring persistent error conditions such as catastrophic network or system errors or persistent congestion at the destination endpoint. If such conditions occur, the system can deem some messages to be undeliverable and return them to their senders. Requests are returned to the requesting endpoint and replies are returned to the replying endpoint; active messages are never dropped silently. Returned messages are received by the endpoints that sent them and behave like ordinary active messages. For instance, they can generate events that can unblock threads waiting for receive-pool transitions from empty to nonempty.

By convention, the first entry in every handler table, denoted handler$_0$, is the *undeliverable message handler*, which as the name suggests, notifies applications of undeliverable messages. Each package can register functions for handler$_0$ that are tailored for their individual needs. For example, returned messages can be dropped, queued at higher layers for later retransmission, or cause the process to abort. The undeliverable message handler is used for both undeliverable requests and replies, and has the following prototype: void handler(int status, op_t opcode, void *argblock). Note that the default handler for handler$_0$ is *abort()*.

The ***status*** value describes why a message was undeliverable. The ***opcode*** argument identifies the type of the returned message, as well as the operation used to send it. ***Opcode*** is one of the following compile-time constants: AM_REQUEST_M, AM_REQUEST_IM, AM_REQUEST_XFER_M AM_REPLY, AM_REPLY_IM, AM_REPLY_XFER_M. The number of arguments passed to the handler$_0$ function is always the same, but the data pointed to by the argblock can differ. For request functions, ***argblock*** points to a structure containing the original arguments. For reply functions, ***argblock*** points to a structure containing the *token* and the original arguments. The table below lists the ***status*** constants and their meanings.

EBADARGS	Arguments to the request or reply function were invalid.
EBADENTRY	The translation-table index selected an unbound table entry.
EBADTAG	The sender's tag did not match the receiving endpoint's tag.
EBADHANDLER	The handler is an invalid index into the receiver's handler table.

Figure 10 Status constants for undeliverable messages and their descriptions

EBADSEGOFF	An offset into the destination-memory segment was invalid.
EBADLENGTH	The bulk transfer length goes beyond a segment's end.
EBADENDPOINT	The destination endpoint did not exist.
ECONGESTION	There was persistent congestion at the destination endpoint.
EUNREACHABLE	The destination endpoint was unreachable due to a serious, likely catastrophic, system or network error.
EREPLYREJECTED	The destination endpoint refused a reply message because the matching request was returned with a status of EUNREACHABLE.

Figure 10 Status constants for undeliverable messages and their descriptions

In all cases besides EUNREACHABLE, the system guarantees that the message produced no events at its destination endpoint. In the case of EUNREACHABLE, the system guarantees to deliver the message at most once, which means that the possibility exists that the message was received and incorporated into the remote computation. It is the responsibility of higher level protocols to detect, rationalize, and handle this case, if necessary. Undeliverable bulk transfer messages can have written portions of their destination-memory regions.

4.3.1 Error model as a signal-detection experiment

Because EUNREACHABLE indicates a serious system problem such as a crashed node, messages returned for this reason are handled differently than those returned for other reasons. Understanding this error condition is particularly important for building highly-available applications. It is useful to frame this situation in terms of a signal-detection experiment and its four possible outcomes, as shown in Figure 11. The setup is as follows. There are two nodes, N_A and N_B, where N_A sends a request to N_B. After the request is sent, N_B is in one of two states (where its state is the "signal" to be detected by node N_A). N_A then measures N_B's signal and decides how to proceed. One possibility is to return the request with status EUNREACHABLE. There are four possible outcomes of this signal-detection experiment.

Case Number	N_A thinks N_B is	State of N_B is
1	DEAD	DEAD
2	ALIVE	ALIVE
3	ALIVE	DEAD
4	DEAD	ALIVE

Figure 11 Semantics of EUNREACHABLE

Cases 1 and 2 are easy to understand, because in both cases, node N_A correctly determined the state of node N_B. Case #1 produces an undeliverable message with status EUNREACHABLE, while case #2 is error free because the reply to the request was delivered. Case 3, represents the false-negative case, in which N_A believes that N_B is alive, but N_B is in fact dead. If N_B remains dead long enough for N_A to correctly determine that N_B is in fact dead, then the request eventually returns to $N_{A,}$ and N_A will update its belief (that N_B is dead). The possibility exists that N_B re-boots while N_A is retrying the request message to N_B. If this happens, N_A's message should be returned, not delivered. Depending on the implementation, the message can be returned because N_A begins sending the request to a now nonexistent endpoint on N_B. Other implementations can tag messages with epoch numbers (e.g., randomly chosen large numbers, or, if the operating system can provide known to be "fresh" epoch numbers, so much the better). Then, when a message arrives for any epoch except the current one, it is dropped. Whenever a requesting node sends a message to a node and it is returned with EUNREACHABLE, the system internally changes the epoch used for future communications with that node. Case 4 is the false-positive case where N_A believes N_B is dead but N_B is in fact alive and communicating. Again, a returned message with status EUNREACHABLE means that communication to the remote node has encountered a catastrophic error. For instance, if N_A requests to N_B and the system returns the request with an error status of EUNREACHABLE, then the system must guarantee with a very high probability that the requestor never sees the corresponding reply should it ever arrive at N_A. The second issue is what happens on the replying node, N_B. The replying application on N_B handles the request and sends back the reply to N_A. But N_A believes that N_B is dead. It is misleading, if not entire incorrect, for N_B to inform the application that its reply message was delivered successfully because the reply should never have been delivered in the first place. The application on N_A can be busy executing quorum or recovery code, assuming N_B is dead. Allowing the reply, which is now "unexpected" to "sneak in" and be delivered, would make writing recovery code very difficult. To prevent this case, the reply must be returned with either an undeliverable status of EREPLYREJECTED or EUNREACHABLE. The status that is returned is implementation-dependent.

4.3.2 Synchronous and asynchronous error detection

The active message transport functions in Appendix A return a value of either AM_OK or AM_ERR_XXX. When errors can be detected synchronously with respect to a caller, the active message functions return immediately with error status. Otherwise, when errors are detected asynchronously with respect to a caller, the system generates undeliverable messages. Whether or not an error can be detected synchronously or asynchronously is implementation dependent, more specifically it depends in part upon what knowledge a sending endpoint has of a destination endpoint's parameters. As one example of the distinction, *AM_Rquest4()* verifies its *requesting_endpoint* argument before sending the message. Yet once it returns, the destination endpoint could be deallocated before the message is delivered, or the destination endpoint's handler table could be resized so that the message then contains an invalid handler-table index.

4.4 Managing congestion at destination endpoints

The function *AM_SetExpectedResources()* allows applications to inform the system of their expected communication requirements. After returning from this function, the system informs the caller if the requested resources were available and were dedicated to the endpoint. The resource requirements are parameterized as the product of the number of distinct endpoints with which a given endpoint intends to communicate and the number of outstanding requests per endpoint.

This resource information enables implementations to adjust flow control and endpoint management protocols such that space in a receive pool is always available for incoming messages, which means that messages need never be returned. Within a set of endpoints, all of which have specified their expected resource requirements, the system never returns messages because of persistent congestion at a destination endpoint (*i.e.*, full receive pools). If the communication resource requirements are unknown or not explicitly specified, or if the application exceeds its stated requirements, then the system continues to operate. However, the system can return messages because of persistent destination endpoint congestion.

The delay between the delivery of an active message and its reception (i.e., its handler being invoked) can be arbitrarily long. The delivery model does not consider receive pools that are consistently full to be an error condition, although the latter can penalize performance. The model does, however, guarantee that a collection of full receive pools will not deadlock communication among unrelated endpoints in the system. The time at which send-pool entries, which logically contain message descriptors, can be reused is implementation-specific and depends on the underlying transport flow control and error detection protocols. For example, send pool storage containing a request could be reused once the corresponding reply is delivered, and send pool storage containing a reply could be reused once the “next” reply is delivered.

5 Endpoint Management with Solaris/Myrinet

The active message interface permits allocation of multiple endpoints per process, a number that can grow to be large, *e.g.,* dozens or more. With many processes using active messages, the total number of endpoints per machine could grow into the hundreds. The Myricom network interfaces are SBUS devices with embedded processors that can operate autonomously from their local SRAMs. Their memory capacity depends on the interface-hardware version and ranges from 128KB to 512KB. Assuming that an endpoint is two 8KB-pages in length and 50% of the memory is used to hold endpoints, then each network interface can hold between 4 and 16 endpoints. These factors combine to raise important questions: how are endpoints on a machine multiplexed onto the single network interface? and how are independence and forward-progress among distinct endpoints guaranteed?

As a simplifying assumption, each machine is assumed to have a single network interface.This is because multiple interfaces reduce the endpoint to interface ratio but introduce new questions about load balancing endpoints across the interfaces or striping operations across multiple interfaces in parallel. The embedded message processor is assumed to be able to deliver messages to endpoints in its local memory faster than it can deliver messages to endpoints in host memory, and that it performs operations on endpoints in its local memory more efficiently than can the host. The final simplifying assumption is that messages can be sent from and delivered to all endpoints independently of their storage in network-interface memory or in host memory. This makes sense if the performance of endpoints in interface memory is expected to be better than that for endpoints in host memory, due to their co-location with the network interface’s embedded message processor. The presence of many endpoints per machine and the stated assumptions about the network interface’s capabilities leads to the following idea: cache the most active or the *hottest* endpoints in the network-interface memory and leave all remaining endpoints in the host’s memory. The address space segment drivers in the Solaris virtual-memory system provide the necessary mechanisms to implement this idea. By assuming that mechanisms exist so that communication can occur independently of an endpoint’s location (in host or network-interface memory), this problem can be further reduced to the problem of finding and implementing policies that manage the working set of endpoints in the network interface.

The following sections describe the design and implementation of a prototype system that implements endpoint paging. They introduce the structures and mechanisms relevant to Solaris process address spaces, the role of Solaris address space segment drivers in managing the endpoint backing storage, and the mechanisms for building endpoint caching policies.

5.1 Solaris address space management

Each Solaris process has a virtual address space that is a collection of address-space segments, each of which is a virtually-contiguous memory region managed by a segment driver. The segment driver manages the virtual-to-physical mappings of their segments. Although the virtual-to-physical mappings (*i.e.,* hardware translations) are transient, the segment driver maintains persistent associations between segments and their backing storage. In the prototype system, every endpoint is a distinct virtual-memory segment in a process’s address space, and a custom segment driver manages all communication endpoint segments per machine.

Figure 12 depicts the address-space structure of a process, an endpoint segment and the endpoint segment driver responsible for its management. The endpoint segment driver is a loadable, Solaris module with predefined entry points. Higher levels of the virtual memory system call to these entry points to perform standard operations on segments, such as allocating a segment, de-allocating a segment, duplicating a segment (for forking), locking and unlocking segment pages in memory, and, most importantly, handling segment pagefaults. The ca-

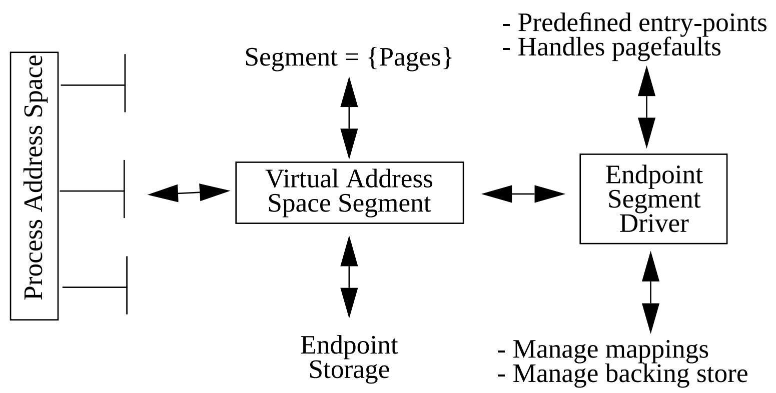

Figure 12 Organization and management of Solaris address spaces

pability to catch and handle segment pagefaults is the key mechanism available from a customized segment driver.

5.2 Managing communication endpoint segments

With this framework, the endpoint segment driver is faced with two tasks: performing basic endpoint management and managing the endpoint cache in the network interface. With respect to the active message interface, the segment driver is used whenever the *AM_AllocateEndpoint()* or *AM_FreeEndpoint()* functions are called, whenever a process with allocated endpoints exits or aborts, or whenever a process faults on an unmapped endpoint. For example, when *AM_FreeEndpoint()* is called, the segment driver frees the backing storage and unloads any mapping to the endpoints. The driver also provides mechanisms to multiplex and cache endpoints on the network-interface card. The mechanisms available for constructing the policies are simple: (1) an endpoint's backing storage can be moved between host and interface memory, (2) the virtual-to-physical hardware translations for endpoints can be loaded or unloaded, and (3) pagefaults can be taken when processors access unmapped endpoint segments. These three, basic mechanisms are sufficient for caching the most active or *best* endpoints, according to some metric, in interface memory.

Figure 13 on page 24 shows the task of mapping endpoint segments onto either kernel memory or network-interface memory. The thick arrows indicate associations of endpoint segments to either interface memory or host memory. These are active mappings, indicating associations for which virtual-to-physical translations exist (*i.e.*, "are loaded"). The thin dotted arrow indicates an association between an endpoint and a region of host memory without an active virtual-to-physical translation. This design has aspects of a demand-paging problem and aspects of a caching problem. Like conventional memory paging where pages move between main memory and secondary storage, here pages move between main memory and network-interface memory. Although the initial prototype first used wired kernel memory for endpoint backing storage, it was subsequently changed to use pageable kernel memory. This change allowed the kernel to transparently to page the backing storage for the inactive endpoints to disk and creates the standard cache-memory-disk hierarchy. Efforts are focusing on understanding and implementing the endpoint caching policies. It is worth noting that when the backing storage for an endpoint is resident in kernel memory, an endpoint pagefault to that backing storage is classified as a "minor" fault, whereas an endpoint pagefault to backing storage that the kernel has paged out to disk is classified as a "major" fault. (The ratio of major to minor endpoint faults in a live system is an interesting parameter to monitor.)

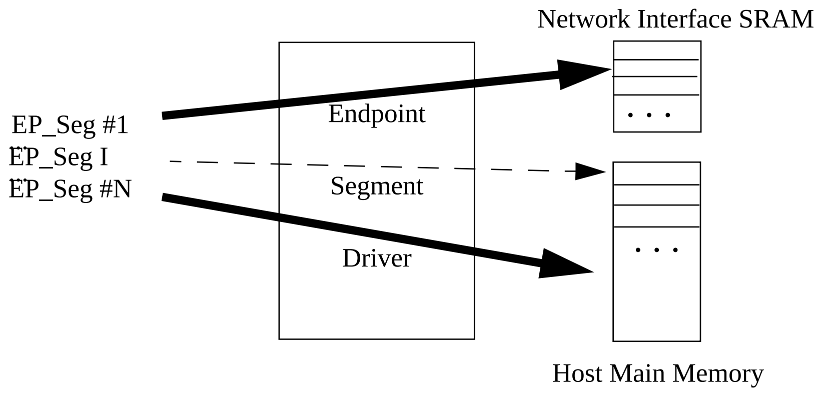

Figure 13 The problem of endpoint caching

5.3 Endpoint segment mechanisms and policies

For any endpoint segment, the segment driver maintains the following invariants: (1) the segment is associated with a region in kernel memory, (2) the segment is associated with a region in interface memory, and (3) the segment is not associated with any region in memory. Case 3 is a transient condition that exists only when a segment is being created or destroyed. Orthogonal to maintaining the associations of segments to their backing storage is maintaining the set of mappings between segments and their backing storage. The virtual-to-physical mappings between segments and their backing store are transient; the driver can unload them at any time. A processor reference to a segment that has no mapping results in a pagefault, which is a synchronous trap taken in the context of a process and which invokes the segment driver's pagefault routine. The faulting process will sleep in the segment driver's pagefault routine until a valid mapping for the segment is loaded, at which time it resumes execution and retries the faulting instruction. It is precisely this ability to manipulate mappings between segments and their backing storage as well as to catch pagefaults that allows a multitude of endpoint management policies to be implemented and evaluated. Whenever a process pagefaults on an endpoint segment, numerous actions can be taken: the process can be left sleeping in the pagefault routine, a hardware translation for the segment can be loaded, or the segment's backing store can be moved between host and interface memory, *etc.*

Policies arise as a result of these mechanisms being applied. For example, one policy is to split the number of physical endpoints on the network interface into two sets. The first set is used to cache the endpoints referenced most recently from the processors. The second set is used to cache a disjoint set of endpoints referenced most recently by arriving network messages. The driver could treat each set as a fully-associative endpoint cache with an LRU replacement policy. A clock-based algorithm, similar to the ones used to maintain physical memory page caches, could be used. (Of course, moving an endpoint's backing storage between host and interface memory requires synchronizing with the interface's message processor.) The best endpoint to evict from the network interface under various circumstances is an interesting research issue. To solve it adequately, policies must be subjected to realistic work-loads, as well as "corner cases" such as when all endpoints have messages to send or when all endpoints are waiting for on message arrival events. By not only managing the memory that contains endpoints but knowing the data structure embedded on those pages, the driver can, for example, distinguish those endpoints with nonempty send or receive pools from those waiting on communications events.

The driver has access to the kernel thread and process scheduling systems. This allows policies to couple communications, endpoint storage management, and process scheduling. Because the virtual-memory mechanisms are easily composed into management policies, the challenge is to evaluate the policy space. For example, the strict LRU policy described above has the potentially unfortunate side-effect of caching endpoints that are referenced infrequently from a processor. On the other hand, it also evicts relatively inactive endpoints even though an application can desire minimal latency and overhead when the endpoint is used. Following the lead of standard virtual-memory management policies, application-controlled endpoint management is a plausible and applicable option because it is unclear, *a priori*, why any one static policy picked by the system will always perform well across a wide variety of different applications.

5.4 Initial experiences with Solaris segment drivers

The first prototype Solaris segment driver for endpoint management used the prepackaged *seg_mapdev* driver, primarily because it existed and supported a restricted version of the more general mechanisms and policies just discussed. It also allows multiple processes to map the same pages of device memory while maintaining the invariant that only one mapping is loaded at any time. Its functionality is thus motivated by the requirements of framebuffers - to allow multiple processes to establish multiple mappings to common device memory. Whenever a process references unmapped memory, a preregistered callback function is invoked, allowing the driver to perform a customized context switch of the device state. While this prepackage segment driver enabled rapid prototyping, it enforced the policy that a process's context *e.g., endpoint*, must be resident on the card before the endpoint can be accessed. Thus, the capability to have endpoints backed by either host or device memory and have active mappings in either case was unavailable. Policies built from this driver would therefore have inadequate control over loading and unloading mappings as well. However, the ability to setup simple test cases where multiple processes mapped the network-interface memory in a demand-paged fashion demonstrated that further development of a fully-featured segment drivers was warranted.

6 Related Work

Network interfaces and their integration with host operating systems are areas of ongoing research, and of the many significant projects in this field, this section discusses five of them. Each one is summarized by a statement of its core concepts, the intended applications or domains of use, and a comparison with the work presented in this document. The five projects are Princeton University's SHRIMP Multicomputer, the University of Arizona's Osiris ATM network adaptor project, Hewlett Packard's Hamlyn project, the SUNMOS system, and the NORMA IPC system. These system are discussed and evaluated based upon varying amounts of published and unpublished works about them, which makes evaluating certain specific or low-level details difficult. From the experience of specifying the organization of and interface to the new active message system, it is often a challenge to specify fully and to define clearly a communications system (particularly one under development) such that readers will understand its various nuances.

6.1 Princeton's SHRIMP multicomputer

The Princeton SHRIMP Multicomputer [33], [34] uses reflective memory as a communications resource. The idea behind reflective memory is to establish associations between virtual-memory regions in different processes and on different processors. Once setup, stores into local pages of reflective memory are automatically "published" to the other associated pages. Moreover, loads from a reflective memory page return the values that are present locally. Reflective memory, however, has weak memory consistency and coherency models, but the required hardware and software is simple and inexpensive, and its simplicity enables low-overhead and high-bandwidth data transport. Despite the use of a Paragon routing backplane, SHRIMP systems are intended to be inexpensive multicomputers. Published papers reveal that its application domain is unclear but presumably includes parallel programs and other, ad hoc, applications.

SHRIMP provides protected network multiprogramming. It uses a collection of trusted third-parties, as well as the operating systems running on each SHRIMP node, to manage the reflective memory mappings. Once mappings are installed, the network interface remains attentive to the network and snoops memory-bus transactions. It also uses page tables to enforce protection and to multiplex traffic on and off the network. The new active message interface shared similar goals of enabling protected network multiprogramming. The common network-interface organization mentioned in this paper suggests that each network interface be viewed as an active agent that performs per-message tag checks. The host operating system assists with caching and paging endpoints on and off the network interface. The endpoint segment driver manages data structures that are equivalent to the network-interface page-tables in SHRIMP.

The SHRIMP Base Library (SBL) document [35] describes how applications control which memory updates generate events and call user-specified handler functions. The notification events are queued in the system, but surprisingly, the SBL document asks programmers to be careful about how long the handlers execute because the event queues have a finite length. This suggests they can overflow and events could be lost. The new active message interface has a different event management mechanism: it projects events to applications using thread-synchronization variables. With binary semaphores, no issues arise with overflows of event queues. Programs avoid using the heavy-weight process signal mechanisms of UNIX for message event handling. The integration of events with threads and endpoints helps reasoning about communications in threaded applications.

Although the SHRIMP conference and journal papers are focused on network-interface designs, they lack discussions about error and fault models and how applications detect and recover from node failures. Support for fault-tolerant or highly-available applications is apparently not a focal point. In contrast, the new active message specification is lengthier than its predecessors because it attempts to provide a model for reasoning about communications “when things go wrong”. Its undeliverable message handling model is one example of this.

The SHRIMP system provides memory-based communications and very low-overhead primitives, although it does not provide atomic memory operations against locations in reflective memory. It is difficult to build an efficient message queue, and the cost of the obvious alternative approach of scanning pages for arriving messages scales poorly (linearly). Queues are fundamental to many communication protocols. With active messages, one can construct queues and other data structures as well as perform remote operations atomically. Both systems face the challenge of providing fast event up-calls. In SHRIMP they are needed to execute handlers and in active messages to post synchronization variables.

6.2 The Arizona Osiris project

The Osiris Project [36] investigates the integration of high-performance ATM host adaptors into the Mach operating system. The principal focus in this work is on integration strategies that deliver low latencies, high bandwidths, and optimal end-to-end performance. A key contribution to this effort is prototyping and evaluating application device channels (ADC’s). ADC’s allow pages of card memory to be mapped into application address spaces. Associated with each end of an application device channel are chains of fbufs, containing messages for sending and buffers for receiving messages. The project focus is apparently on network interface and operating systems research. No specific mention of target applications is made, although Osiris serves as an efficient ATM-based infrastructure for distributed system applications.

The Osiris adaptor card not only manages segmentation and reassembly of large messages, but it also determines when to interrupt the host for message arrival events. It also supports protected data transport since the host virtual memory system managed device page mappings via ADC’s. The memory resources of the network interface, their containment of distinct send and receive structures, and the capacity of the host to map individual ADC’s enable protected network multiprogramming.

The system, however, lacks flexible control over event generation because the system provides a single “message arrival” event. These events either wake threads in the kernel ADC driver or user-level task threads, waiting to process messages. The published paper was unclear if the event upcall was into a Mach kernel object or into a Mach task. The naming and protection models used ATM VCI’s but the paper did not address how the system scales once the VCI space is exhausted; message queues are not discussed. The network interface organization and its integration into the host operating system supports overlapping of communication and computation, with an emphasis on off-loading computation and interrupt handling burdens from the host. The new active message specification has more general naming, protection, and event models. For example, rather than associating a chain of fbufs with an ADC, it associates a single application-specified virtual address region with an endpoint. The active message interface provides applications more control over *which* memory buffer receives which message.

With respect to the endpoint paging prototype system, it effectively reinvents ADC's and extends them to support more channels (or endpoints) that the physical memory on the network adaptor card alone permits. In other words, the endpoint paging system virtualizes the network-interface memory resources, and therefore it's possible to support more endpoints than the physical interface memory capacity alone allows. The design clearly extends ADC's by viewing the network-interface memory as a cache of endpoints, where host memory can be used as a backing storage for all but the hottest endpoints. The implementation is now at a stage where the feasibility of paged "application device channels" can be demonstrated. Additionally, endpoints in host memory can send and receive messages, though with less performance compared to endpoints cached in the network interface.

6.3 Hewlett Packard's Hamlyn project

The HP Hamlyn [37] Project is building a multicomputer network interface, which is currently being emulated on the Myrinet LANai network interface [9]; a set of communication protocols, RATS, provide communication services. The Hamlyn paper discusses both the network interface's macro-architecture and the communication protocols that use the interface. Hamlyn supports direct application - network interface interactions that bypass the OS, zero-copy implementations of bulk data transfers, protected network multiprogramming, and message delivery events. The RATS protocols support remote writes, remote writes-with-notification, reliable datagram transport with no expected reply, reliable datagram transport with an expected reply, remote read, reliable byte-streams, and initial request response operations (to make initial "unexpected" contact with remote parties). The paper indicates that both the network interface and the RATS protocols should support general-purpose communications within the multicomputer.

In Hamlyn, an application has one "send terminus" and multiple "send areas". A send area is a collection of data segments mapped into their virtual address spaces. The Hamlyn model is an explicit sender-specified one-way memory copy. The sender specifies the receiver's send area, key, and offset into the array of data segments. The physical security of the target networks enables a protection system using 32-bit "keys": senders can write into a receiver's send area if the key in the sender's message matches the key in the sender-specified send area segment. Message arrivals can cause events to be enqueued into a *per-process* notification queue; message arrivals interrupt sleeping destination processes only. As in SHRIMP, the notification queues can apparently overflow with unspecified consequences. Interrupts are directed at processes, not threads. In the new active message specification, the precision with which events are delivered is more precise.

Hamlyn assumes a network with low transmission error rates. This contrasts somewhat with the design philosophy behind the new active message interface which acknowledges that even if network fabrics were 100% reliable, the network hosts certainly are not. The error and message delivery model are required to provide meaningful communication facilities in a real-world system. In the Hamlyn paper, there was no mention, however, of error and fault models or how application detects the failure of a remote node, let alone how it should recover. There was also no mention of building message queues or how either the interface or RATS protocols support their efficient construction. However, the low-level support for scatter, gather and broadcast operations in Hamlyn could be quite useful.

In general, the relationship between Hamlyn and RATS is confusing. Hamlyn appears to be a network-interface macro-architecture and RATS appears to be a higher-level set of protocols optimized for it. The new active message specification is at an intermediate level of abstraction. It walks the fine line of providing primitives and simple abstractions that would be efficiently implementable on hardware, such as Hamlyn, and of providing primitives that efficiently compose into the protocols, such as RATS. Hamlyn, RATS, and this document speak at different levels of abstraction.

Lastly, the paper is incorrect in stating that the active message model requires application gang scheduling. Gang scheduling is a performance optimization that will benefit both latency-sensitive Hamlyn and active message applications alike.

6.4 SUNMOS

SUNMOS [38] divides a set of computing nodes on a high-performance network into ones running a fully-general operating system, as well as ones running a bare-bones micro kernel. It explores the potential performance of a specialized micro-kernel operating system. Besides specializing the function of specific nodes, other SUNMOS specializations include: removing virtual-memory facilities, optimizing for the execution of a single parallel process on dedicated hardware, and providing a *simple* message-passing library with send and receive operations on un-tagged messages. It's focus does not obviously include more common or general-purpose applications such as clients and servers. These applications would benefit from high performance communications as well, although supporting them would require undoing some of the specializations for parallel programs.

SUNMOS does not provide an infrastructure with optimized communication for mainstream applications such as client/server applications or data base applications that require per-node multiprogramming. A major shortcoming of SUNMOS is its specialization to space-shared parallel programs and its lack of an adequate scalable I/O system. To its credit, it does provides parallel applications with direct network access and nearly optimal end-to-end performance at the expense of standard distributed and parallel system functionality.

The PUMA message passing library provides traditional send and receive functions, but lacks buffer tags and traditional tag matching functions. Thus, applications cannot control which messages are received into which buffers. In this respect, the model of buffer management is similar to Osiris' use of fbufs chains. It also resembles remote queue models such as [26]. The new active message interface specifically addresses the need for composing independent software packages into programs and maintaining their independence, although removing tags from the message passing library complicates this issue.

SUNMOS and the new domain of active messages are opposites in most respects except two: both support direct application - network interface interactions that bypass the OS, and both strive for the best possible end-to-end performance. However, their problem domains are significantly different, largely because the intent of the new active message specification is to provide a portable, general-purpose communications API that supports a much broader range of applications than just parallel programs.

6.5 NORMA IPC

NORMA (NO Remote Memory Access) IPC [39] is an extension of the Mach IPC mechanisms for multicomputers. It is a kernel-based communications facility for protected, multi-programmed, task-to-task communications. Mach *tasks* communicate using *ports* that have distinct *send rights* and *receive rights*. Ports have a private, kernel-managed memory area used to maintain lists of and rights to all other ports with which a port can communicate. Messages are passed between ports providing the sender and receive hold the appropriate send-and-receive rights, respectively. One task can receive messages from a port; many can send messages to a port. The transport protocols are traditional: they involve copying data from the sending task down into the kernel, across the network, and up into the receiving task. The IPC facilities are intended to support general-purpose communications such as datagram or remote procedure call (RPC) systems. Because the focus is on functionality and not performance, issues such as direct task - network interface interactions are not discussed.

NORMA IPC assumes an initial, static, domain of a multicomputer. It lacks any mention of or emphasis on direct application - network interface interaction. All communications are mediated by the kernel which involves traps and multiple protection boundary crossing and data copying. NORMA IPC and active messages target different design spaces. Mach IPC, of which NORMA IPC is an extension for multicomputers, is used for implementing RPC's and bulk transfers. RPC's are significantly different from the request and reply operations in the new active message interface. It can be argued that both NORMA IPC mechanisms and active messages use capabilities (though port rights are more "traditional" capabilities). The NORMA IPC system is certainly more complicated. For example, it maintains reference counters to capabilities and can notify receivers when no other ports exist with send rights. The active message protection model follows V and Amoeba, where object identifiers and tags form user-level capabilities (without reference counts).

6.6 Summary

From the discussion of these five related projects, it is clear that active messages have overlapping aspects with all of them. With SHRIMP, Osiris and Hamlyn, for instance, the drive for simplicity and performance stems from direct application - network interface interactions. Another area of overlap with Osiris is the focus on integrating the network interface with the host operating system, and the virtualization of the interface's physical resources. The new active message interface is as general as that of the NORMA IPC system. Moreover, all of these systems faced with a common set of issues: naming, protection, error models, message delivery models, events and notification. The active message specification is far more general than these systems with respect to its position on naming, more forward-looking in its treatment of threads and events, and more realistic because of its explicit message delivery model and handling of undeliverable messages. It is fortunate that active message implementations can borrow from the successes of these related systems and yet can attempt to avoid their pitfalls.

7 Summary

Networks continue to evolve with increasing link bandwidths and decreasing switch latencies. Network interfaces also continue to develope and to exhibit many diverse organizations. At the present time, there is clustering in their design space around organizations with autonomous, embedded message processors (or controllers) that operate from local memories and that support DMA and PIO interfaces for sending and receiving messages. As link bandwidths increase, network-interface resources, their organizations, and their integration into a host become increasing important. The software overheads paid to send or to receive a message directly determine how effectively an application uses the available network performance. For low-latency short message communication, direct application-network interface interaction is essential. For optimal short message performance, both the sending and receiving applications should be co-scheduled. For optimal bulk data transfers, for applications with "enough" locality, intermediate data copies and movement should be avoided. Zero-copy bulk transfer implementations require that the necessary memory resources for the source and destination be pre-initialized and readied for use in DMA transfers. Thus, the memory resources must be similarly co-scheduled for optimal bulk transfer performance, though this is independent of the co-scheduling the associated *processes*.

Motivated by these emerging networks and a common organization for contemporary network interfaces, this document describes a new, portable, and general-purpose active message communications interface. It used previous active message interfaces as a starting point, given their successes in constructing high-performance application, communication libraries and protocols. These previous interface implementations were tailored to the peculiar, predetermined environments of commercial MPPs. For example, some required gang-scheduling of parallel processes that use them and others required handlers to execute within a predetermined length of time. Such implementation artifacts, however, no longer influence the interface semantics. Additionally, the success at unifying previous active message interfaces into the Generic Active Messages API demonstrated that active messages are a portable and efficient communication abstraction. These factors led us to believe that the time is right to bring a new specification of active messages into the mainstream.

But bringing active messages into the mainstream requires evolving previous naming and protection models, error and message delivery models, and integrating active messages, threads, and events. The naming model enables client/server and peer/peer communication, where the name space of communicating entities is dynamic. The protection model is simple and provides a high-degree of protection, given well-chosen tags from a large sparse tag space. The enhanced message delivery and fault model specifies the conditions under which messages become undeliverable and then how they are handled. Returning all undeliverable messages to their senders allows fault-tolerant and high-available applications, such as file systems, operating systems, and information servers to be constructed. The ability of the network interfaces to exert control over the physical network supports handling undeliverable messages in an intelligent and useful manner. Use of the interface in multi-threaded programs and on multiprocessor machines, for instance, led to the integration of communication events and notifications with thread-synchronization mechanisms, rather than clumsy process signal mechanisms or ad hoc interrupt systems. This integration enables event-driven execution of multi-threaded applications, and permits implementations to exploit local processor parallelism when sending, receiving, and handling active messages.

Virtualization of the physical network resources from applications to their endpoints is a key requirement. The success of the prototype endpoint paging system and the ease with which we were able to use the Solaris address space segment drivers to experiment with solutions bolstered confidence that a complete solution is possible. Furthermore, it became clear from the initial studies that while the mechanisms for managing endpoint backing storage and endpoint mappings are readily available, the space of endpoint caching and scheduling policies needs to be thoroughly investigated. The ability of the host to send and receive messages for endpoints in main memory, and the integration of the network, the virtual memory system, and the process scheduling system makes possible a host of novel implementations and studies.

Finally, appendices A and B specify the new interface. They addresses previous shortcomings with minimal new, but necessary, functionality. The specification generalizes the active message paradigm yet retains their simplicity and architectural integrity. Although tempted to add new functionality and explore alternative communication models, the interface stays with an active message model. This is not a judgement of the other models, merely a consequence of deciding that moving current active message functionality and performance into the mainstream has the highest priority.

7.1 Current status

For the Berkeley NOW project[5], design and implementation of the new active message interface is proceeding using UltraSPARC-1 workstations running Solaris 2.5 with a Myrinet network. A *reference* implementation of the specification using UDP sockets has been completed. An independent effort is underway at Cornell that

5. The Berkeley NOW system is a collection of workstations on a low-latency and high-bandwidth switched network. Participation in the NOW grants benefits and privileges to its members. For example, the collective disk, memory, and cup resources of the system are available to members and their serial and parallel programs. Membership in the NOW relinquishes complete local control over a workstation and permits local machine resources to be managed by and used for the benefit of the larger collective. The constituent workstations can be personal computers (PCs); the distinguishing feature [8] is the presence of a full-functionality, time-shared operating system such as UNIX. The Berkeley NOW uses single and multiprocessor SPARC workstations running Solaris 2.5. The workstations reside in machine rooms and in laboratory workspaces. At a high-level, the machines have homogeneous operating systems and network interface implementations. However, at a lower-level, the machines have heterogeneous instruction set architectures and memory system organization. The Berkeley NOW uses the Myrinet network and LANai network interface manufactured by Myricom, Inc.

uses SPARC workstations running SunOS with an ATM network. Working with colleagues at Cornell greatly increased confidence that the new interface will perform well in systems with the previously discussed network-interface organizations.

7.2 Future work

While formulating the new specification, new features, communication models, *etc.*, were discussed but intentionally postponed. Experiences from implementations of the current interface will help evaluate these proposed extensions:

1. Exposing receive pools with a remote queue interface.

Future versions of the interface will expose the send- and receive-pools to applications via a remote-queue interface. This will allow applications to customize the interpretation of send- and receive-pool entries. Currently, the active message model tacitly imposes one interpretation upon the send and receive entries, *e.g.*, a send-pool entry contains a command, a local-endpoint address, a translation-table index, an endpoint tag, an active message handler-table index, and a fixed number of arguments, and so forth. Exposing the entries will allow applications to specialize the contents and interpretation of entries. Others [25], [26] advocate and have implemented atomic remote message queue functions. But for the time being, equivalent functionality can be constructed using the current active message operations.

2. Using alternative schemes to authenticate messages and protect endpoints.

Given well-chosen tag values from the 64-bit tag space, the current endpoint-protection mechanism provides a nontrivial degree of protection and authentication in a physically-secure network. Although additional security mechanisms could be added, researching physically-secure networks was not of highest priority. There was a lack of evidence that system security overall would improve with minor changes to lightweight mechanisms; endpoint tags are adequate for current purposes. Similar protection mechanisms using were successfully used in V [21] and Amoeba [22].

3. Shared memory support for endpoint virtual-memory segments.

Consideration has been given to integrating the BlizzardS [27] shared memory technology from the University of Wisconsin with endpoint virtual-memory segments. The idea here is to allow groups of endpoints to keep specially-tagged endpoint virtual-memory segments cache-coherent. Similarly, it appears at first glance, that the C Region Library [28] could be easily incorporated into the exiting structures of the endpoints and associated memory segments.

4. Provide unreliable versions of the request and reply transport functions.

In certain application contexts (“multimedia” is the fashionable buzzword), the delivery guarantees of the current interface may be unnecessarily strong. Relaxing the undeliverable message model to allow messages to be dropped without notification may be useful or may provide better performance. However, lacking experimental evidence to support this functionality and its performance benefits with respect to active messages, it was postponed.

5. Specialize request and reply functions for passing double-precision floating point and 64-bit integer values.

On many architectures, active message functions can be specialized for composing messages and invoking their handlers, especially when passing double-precision floating point values or 64-bit integer values. The current interface requires pulling such values apart and transporting them as 32-bit integers. Since including these functions would involve only additions to the interface, they can be added at any time. Without a thoughtful study, the simplistic approach of replicating and then specializing existing functions for double-precision floating point values and for 64-bit integers could easily result in an unfortunate interface bloat.

6. Using fbufs [29] or similar functionality for managing the storage for bulk transfers.

The current specification takes the position that applications should be able to send from arbitrary regions of their address space and should be able to restrict the portion of their address space exported to other endpoints with the appropriate tag. The virtual-memory segment used for receiving bulk transfers is itself an application-specified memory region. If the memory used for bulk transfers is managed *specially*, then its use in bulk transfer operations can be optimized. For example, the memory can be specially allocated, managed, or pre-initialized for DMA operations. The introduction of special communications memory and its impact on higher-level protocols and applications is worthy of further investigation. Trade-offs between changing programming models to accommodate special memory management functions and using the much improved high-bandwidth memory-copying facilities of modern workstations need to be investigated.

8 Acknowledgments

Many people contributed to the development and the refinement of the new active message interface specification. I would like to thank the following people for their time, energy, and valuable input: David Culler, Eric Brewer, Tom Anderson, Dave Patterson, Eric Anderson, Mike Dahlin, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Steve Lumetta, Rich Martin, Drew Roselli, Randy Wang, Chad Yoshikawa at the University of California at Berkeley; Thorsten von Eicken and Anindya Basu at Cornell University; Adam Greenberg at Quadruped Systems, Inc.; Kristin Wright at Geoworks; and Steven Reinhardt at the University at Wisconsin at Madison. Bill Nesheim at Sun Microsystems took time out from his busy schedule to discuss the integration of the Myricom network interface into Solaris and explain various and sundry details of the Solaris virtual memory system. I would also like to thank Brent Chun for his design and implementation of the reference implementation of the new active message specification using UDP, and for his demonstration of a fault-tolerant application built using it. His efforts turned up several inconsistencies and bugs that had remained invisible for months. Jim Gray and Joe Barrera at Microsoft Research Labs also provided many useful comments on the interface. Lastly, I would like to thank Tom Philippi for helping convert this manuscript into something approximating English.

This research is supported in part by ARPA grant F30602-95-C-0014, the California State Micro Program, and Professor David Culler's NSF Presidential Faculty Fellowship CCR-9253705. Computational and experimental infrastructure resources are supported in part by NSF Infrastructure Grant CDA-8722788.

9 References

[1] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992., Gold Coast, Qld., Australia, pp. 256-266.

[2] L. Tucker and A. Mainwaring, "CMMD: Active messages on the CM-5", Parallel Computing, August 1994, vol.20, (no.4):481-496.

[3] R. Martin, "HPAM: An Active Message Layer for a Network of HP Workstations", In *Proceedings of Hot Interconnects II*, August 1994.

[4] L. Liu, "An Evaluation of the Intel Paragon Communication Architecture", M.S. Project Report, Computer Science Division, University of California at Berkeley, July 1995.

[5] D. Culler, K. Keeton. L. Liu, A. Mainwaring, R. Martin, S. Rodrigues, and K. Wright, "Generic Active Message Specification", Computer Science Division, University of California at Berkeley, White Paper, August 1994.

[6] K. Schauser and C. Scheiman, "Experiences with Active Messages on the Meiko CS-2", In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[7] T. von Eicken, A. Basu, and V. Buch, “Low-latency communication over ATM networks using Active Messages”, In *Proceedings of Hot Interconnects II*, August 1994.

[8] T. Anderson, D. Culler, and D. Patterson, “A Case for NOW (Networks of Workstations)”, IEEE Micro, February 1995, vol.15, (no.1):54-64.

[9] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and Wen-King Su, “Myrinet: A Gigabit-per-Second Local Area Network”, IEEE Micro, February 1995, vol.15, (no.1):29-36.

[10] R. Horst, “TNet: A Reliable System Area Network”, IEEE Micro, February 1995, vol.15, (no.1):37-45.

[11] J. D. Tygar and Bennet S. Yee, “Strongbox: A System for Self-Securing Programs”, CMU Computer Science: A 25th Anniversary Commemorative, ed. R. Rashid (New York: ACM Press, 1991).

[12] Bennet Yee, “Using Secure Coprocessors”, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1994.

[13] O. Babaoglu and A. Schiper, “On Group Communication in Large-Scale Distributed Systems”, *Operating Systems Review*, January 1995, vol.29, (no.1):62-67.

[14] R. van Renesse, T. M. Hickey, and K. P. Birman, “Design and Performance of Horus: A Lightweight Group Communication System”, Technical Report TR94-1442, Computer Science Department, Cornell Univ., August 1994.

[15] Internet Protocol. DARPA Internet Program. Protocol Specification. Prepared for Defense Advanced Research Projects Agency, Information Processing Techniques Office by the Information Science Institute, University of Southern California. September 1981, pp. 6-7.

[16] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard”, Computer Science Department Technical Report CS-94-320, University of Tennessee, May 1994.

[17] P. Bangalore, N. Doss, and A. Skjellum, “MPI++: Issues and Features”, Department of Computer Science, Mississippi State University, White Papers, March 1994.

[18] A. Skjellum, N. Doss. K. Viswanathan, A. Chowdappa, and P. Bangalore, “Extending the Message Passing Interface (MPI)”, Department of Computer Science, Mississippi State University, White Paper, March 1994.

[19] A. Skjellum, N. Doss, and K. Viswanathan, “Inter-communicator Extensions to MPI in the MPIX (MPI eXtension) Library”, Submitted to ICAE Journal special issue on Distributed Computing, July 1994.

[20] A. Skjellum, S. Smith, N. Doss, A. Leung, and M. Morari, “The Design and Evolution of Zipcode”, *Parallel Computing*, April 1994, vol.20, (no.4):565-96.

[21] Marvin Theimer, *personal communication*, 13 February 1995.

[22] A. S. Tanenbaum, S. J. Jullender, and R. van Renesse, “Using Sparse Capabilities in a Distributed Operating System”, Technical Report, Department of Mathematics and Computer Science, Vrije Universiteit.

[23] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, “SunOS 5.0 Multi-thread Architecture”, In *Proceedings of the Winter 1991 USENIX Conference*, January 1991.

[24] “pthreads and Solaris threads: A Comparison of two user-level threads APIs”, SunSoft Corporation, 1994. On-line at http://www.Sun.COM:80/cgi-bin/show?sunsoft/Developer-products/sig/threads/posix.html.

[25] A. Basu, V. Buch, W. Vogels, and T. von Eicken, “U-Net: A User-Level Network Interface for Parallel and Distributed Computing”, In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.

[26] E. Brewer, F. Chong, L. Liu, J. Kubiatowicz, and S. Sharma, “Remote Queues: Exposing Network Queues for Atomicity and Optimization”, In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1995.

[27] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, D. A. Wood, “Fine-grain Access Control for Distributed Shared Memory”, In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[28] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach, “CRL: High-Performance All-Software Distributed Shared Memory”, Technical Report LCS-TM-517, MIT Laboratory for Computer Science, March 1995.

[29] P. Druschel and L. Peterson, “Fbufs: A High-Bandwidth Cross-Domain Transfer Facility”, In *Proceedings of the 14th Symposium on Operating Systems Principles*, December 1993.

[30] K. Keeton, T. Anderson, and D. Patterson, “LogP Quantified: The Case for Low-Overhead Local Area Networks”, In *Proceedings of Hot Interconnects III*, August 1995.

[31] J. Beecroft, M. Hornewoord, and M. McLaren, “Meiko CS-2 Interconnect Elan-Elite Design”, *Parallel Computing*, November 1994, vol.20, (no.10-11):1627-38.

[32] Fore Systems, Inc., “200-Series ATM Adapter - Design and Architecture”, January 1994.

[33] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, *et. al*,. “Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer”, In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[34] M. Blumrich, C. Dubnicki, E. Felten, K. Li and M. Mesarina, “Two Virtual Memory Mapped Network Interface Designs”, In *Proceedings of Hot Interconnects II,* August 1994.

[35] “The SBL Programming Model”, SRIMP Project Document, Department of Computer Science, Princeton University. On-line at http://www.cs.princeton.edu/shrimp/htMan/SBLmodel.html.

[36] P. Druschel, L. Peterson, and B. Davie, “Experiences with a High-Speed Network Adaptor: A Software Perspective”, In *Proceedings of the SIGCOMM ‘94 Symposium,* August 1994.

[37] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes, “Hamlyn: a high-performance network interface with sender-based memory management”, Technical Report HPL-95-86, Hewlett-Packard Laboratories, July 1995.

[38] B. Traversat, “Distributed-Memory OS for Highly Parallel Systems: Experiences and Lessons from Paragon
OSF/1 and SUNMOS”, Technical Report RND-94-015, NASA Ames Research Center, September 1994.

[39] B. Bryant, A. Langerman, S. Sears and D. Black, “A Task-to-Task Communication System for Multicomputer
Systems”, Draft Technical Report, OSF Research Institute, October 1993.

Appendix A Active Message API

This appendix defines the interface for the active message transport functions.

A.1 Data types and constants

Figure 14 defines the data types for the interface.

Data Type	Description
int	32-bit integer
ep_t	endpoint object
eb_t	endpoint bundle object
en_t	an opaque global-endpoint name
tag_t	endpoint tag (a 64-bit integer)
handler_t	handler-table index

Figure 14 Interface data types and their descriptions

Figure 14 defines the endpoint, bundle, and transport function constants for the interface.

Constant	Description
AM_ALL	deliver all messages to endpoint
AM_NONE	deliver no messages to endpoint
AM_PAR	concurrent bundle/endpoint access
AM_SEQ	sequential bundle/endpoint access

Figure 15 Interface constants and their descriptions

Figure 14 defines the function return values and error constants for the interface.

Data Type	Description
AM_OK	function completed successfully
AM_ERR_NOT_INIT	active message layer not initialized
AM_ERR_BAD_ARG	invalid function parameter passed
AM_ERR_RESOURCE	problem with requested resource

Figure 16 Function return values and their descriptions

AM_ERR_NOT_SENT	synchronous message not sent
AM_ERR_IN_USE	resource currently in use

Figure 16 Function return values and their descriptions

Note: The active message transport functions are parameterized by ***M***, the number of integers argument passed to the active message handler. All implementations will provide at least two versions of all functions with **M=4** and **M=8**. Rather than repeat the active message function and handler semantics for every transport function, readers should refer to Section 4 for those details.

Note: In general, all *transport* functions return integer value of AM_OK if no problems with the given parameters are found synchronously with respect to the calling computation and AM_ERR_XXX otherwise. Subsequent errors may arise asynchronously with respect to the calling computation and result in undeliverable messages being returned to their senders. All *other* functions return integer values of AM_OK if no problems with the given function parameters are found and the operation completes successfully and AM_ERR_XXX otherwise. These functions execute synchronously with respect to the calling computation and thus asynchronous error detection issues is a non-issue.

A.2 Function: AM_Init

int rval = AM_Init()

AM_Init initializes the active message layer and should be called before using any interface function. Multiple calls return success. The function returns AM_OK if the active message layer was successfully initialized and AM_ERR_NOT_INIT otherwise.

A.3 Function: AM_Terminate

int rval = AM_Terminate()

This function should be called when finished using active message. It cleans up and releases associated system resources. *AM_Terminate* returns AM_OK if all active message layer state is released and AM_ERR_NOT_INIT otherwise.

A.4 Function: AM_RequestM

```
int rval = AM_RequestM(
    ep_t request_endpoint, /* endpoint sending request */
    int reply_endpoint,    /* endpoint sending reply */
    handler_t handler,     /* index into destination endpoint's handler table */
    int a_0,..., int arg_M-1)
```

AM_RequestM returns control to the calling thread of computation after sending the request message. Upon receipt, the receiver invokes the active message handler function with the *M* integer arguments. The request handler's prototype is: *void handler(void *token, int* arg_{0},..., *int* arg_{M-1}).

A.5 Function: AM_RequestIM

```
int rval = AM_RequestIM(
      ep_t request_endpoint,/* endpoint sending request */
      int reply_endpoint,      /* endpoint sending reply */
      handler_t handler,       /* index into destination endpoint's handler table */
      void *source_addr,
      int nbytes,
      int a0,..., int argM-1)
```

AM_RequestIM returns control to the calling thread of computation after sending *nbytes* of contiguous data and then the associated request. The active message is logically delivered after the data transfer finishes. Upon receipt, the receiver invokes the handler function with a pointer to storage containing the data, the number of data bytes transferred, and the ***M*** integer arguments. The dynamic scope of the storage is the same as the dynamic scope of the handler. The data should be copied if it is needed beyond this scope. The value of *nbytes* must be no larger than the value returned by *AM_MaxMedium.* The request handler's prototype is: *void handler(void *token, void *buf, int nbytes, int* arg_{0}*,..., int* arg_{M-1}*).*

A.6 Function: AM_RequestXferM

```
int rval = AM_RequestXferM(
      ep_t request_endpoint,/* endpoint sending request */
      int reply_endpoint,      /* endpoint sending reply */
      int dest_offset,
      handler_t handler,       /* index into destination endpoint's handler table */
      void *source_addr,
      int nbytes,
      int a0,..., int argM-1)
```

AM_RequestXferM returns control to the calling thread of computation after sending *nbytes* of contiguous data and then the associated request. The data transfer is offset into the destination endpoint's virtual-memory segment by *dest_offset* bytes. The active message is logically delivered after the bulk transfer finishes. Upon receipt, the receiver invokes the handler function with a pointer into the endpoint virtual-memory segment, the number of data bytes transferred, and the *M* integer arguments. The value of *nbytes* must be no larger than the value returned by *AM_MaxLong.* The request handler's prototype is: *void handler(void *token, void *buf, int nbytes, int* arg_{0}*,..., int* arg_{M-1}*).*

A.7 Function: AM_RequestXferAsyncM

```
int rval = AM_RequestXferAsyncM(
      ep_t request_endpoint,/* endpoint sending request */
      int reply_endpoint,      /* endpoint sending reply */
      int dest_offset,
      handler_t handler,       /* index into destination endpoint's handler table */
      void *source_addr,
      int nbytes,
      int a0,..., int argM-1)
```

AM_RequestXferAsyncM attempts to post the request operation with the communication system. If the communication system accepts the message for transmission, the function returns immediately with a status value of

AM_OK and otherwise with status value of AM_ERR_XXX. If accepted, the caller must not modify the source memory until it executes the matching reply handler. Some implementations permit multiple *AM_RequestXferAsyncM* operations to be posted and overlapped with local computation. If not accepted the caller must retry, and otherwise this function is similar to *AM_RequestXferM*.

A.8 Function: AM_ReplyM

int rval = AM_ReplyM(
void *token,
handler_t handler, /* index into destination endpoint's handler table */
int a_{0},..., int arg_{M-1})

AM_ReplyM returns control to the calling thread of computation after sending the reply message to the requesting endpoint responsible for the particular invocation of the request handler. Upon receipt, the requester invokes the handler function with the *M* integer arguments. The reply handler's prototype is: *void handler(void *token, int arg_{0},..., int arg_{M-1}).*

A.9 Function: AM_ReplyIM

int rval = AM_ReplyIM(
void *token,
handler_t handler, /* index into destination endpoint's handler table */
void *source_addr,
int nbytes,
int a_{0},..., int arg_{M-1})

AM_ReplyIM returns control to the calling thread of computation after sending *nbytes* of contiguous data and then the associated reply message to the requesting endpoint responsible for the particular invocation of the request handler. The active message is logically delivered after the data transfer finishes. Upon receipt, the requester invokes the handler function with a pointer to storage containing the transferred data, the number of data bytes transferred, and the ***M*** integer arguments. The dynamic scope of the storage is the same as the dynamic scope of the handler. The data should be copied if it is needed beyond this scope. *nbytes* must be no larger than the value returned by *AM_MaxMedium*. The reply handler's prototype is: *void handler(void *token, void *buf, int nbytes, int arg_{0},..., int arg_{M-1}).*

A.10 Function: AM_ReplyXferM

int rval = AM_ReplyXferM(
void *token,
int dest_offset,
handler_t handler, /* index into destination endpoint's handler table */
void *source_addr,
int nbytes,
int a_{0},..., int arg_{M-1})

AM_ReplyXferM returns control to the calling thread of computation after sending *nbytes* of contiguous data and then the associated reply message to the requesting endpoint responsible for the particular invocation of the request handler. The data transfer is offset into the destination endpoint's virtual-memory segment by *dest_offset* bytes. The active message is logically delivered after the bulk transfer finishes. Upon receipt, the requester invokes the handler function with a pointer into the endpoint virtual-memory segment, the number of data bytes transferred, and the *M* integer arguments. The value of *nbytes* must be no larger than the value returned by *AM_MaxLong*. The reply handler's prototype is: *void handler(void *token, void *buf, int nbytes, int arg$_{0}$,..., int arg$_{M-1}$).*

A.11 Function: AM_GetXferM

```
int rval = AM_GetXferM(
      ep_t request_endpoint,/* endpoint issuing get */
      int reply_endpoint,     /* endpoint sending data */
      int source_offset,      /* offset into remote seg */
      handler_t handler,      /* index into destination endpoint's handler table */
      int dest_offset,
      int nbytes,
      int a0,..., int argM-1)
```

AM_GetXferM attempts to post the operation with the communication system. If the communication system accepts the message for transmission, the function returns immediately with a status value of AM_OK and otherwise with status value of AM_ERR_XXX. If not accepted the caller must retry. *AM_GetXferM* retrieves *nbytes* of contiguous data from the replying endpoint's memory segment offset by *source_offset*. The returning data is written into the caller's memory segment offset by *dest_offset.* The active message handler is invoked after the bulk transfer finishes with a pointer to the data, the number of data bytes transferred, and the *M* integer arguments. *nbytes* must be no larger than the value returned by *AM_MaxLong.* The get handler's prototype is: *void handler(void *token, void *buf, int nbytes, int arg$_{0}$,..., int arg$_{M-1}$).*

This operation should be thought of as a primitive composition of a *request* and a *reply_xfer.* It has the same calling restrictions as a request. The handler executes on the *requesting* node, there is no explicit event associated with the operation on the *replying* node, and the handler has the semantics of a reply handler. It is the application's responsibility to manage the source and destination storage and to not modify either until the operation completes.

A.12 Function: AM_Poll

int rval = AM_Poll(eb_t bundle)

AM_Poll services incoming active messages from all endpoints in the specified bundle. The function returns after receiving an implementation-specific number of messages. This prevents a calling thread from being pinned by a long stream of incoming messages. A correctly written program cannot distinguish this behavior from a gap in the stream of arriving messages.

A.13 Function: AM_MaxShort

int rval = AM_MaxShort()

AM_MaxShort returns the maximum number of integer handler arguments, *i.e.* **M**, that can be passed using request or reply operations. This value is guaranteed to be at least eight (words).

A.14 Function: AM_MaxMedium

int rval = AM_MaxMedium()

AM_MaxMedium returns the maximum transmission unit for medium active messages in bytes. This value is guaranteed to be at least 512 bytes across all implementations.

A.15 Function: AM_MaxLong

int rval = AM_MaxLong()

AM_MaxLong returns the maximum transmission unit for active xfer and get operations in bytes. This value is guaranteed to be at least 8192 bytes across all implementations.

Appendix B Endpoint and Bundle API

This sections defines the interface to the endpoint and bundle management functions. All endpoint and bundle operations are atomic with respect to communications with dependencies on the relevant endpoints or bundles. Some examples of such operations are adding endpoints, moving endpoints, deleting endpoints, and changing endpoint parameters.

B.1 Function: AM_AllocateBundle

int rval = AM_AllocateBundle(int type, eb_t *endb, en_t *endpoint_name)

AM_AllocateBundle creates an endpoint bundle of specified type. The *type* is either AM_SEQ or AM_PAR where AM_SEQ indicates that the application *promises* that at most one of its threads of computation will access the bundle at any time and AM_PAR indicates that application *intends* to have multiple application threads concurrently accessing the bundle. If successful *endb* contains the address of the new bundle and *endpoint_name* points to a globally-unique endpoint name. The function returns AM_OK if the bundle was created and AM_ERR_XXX otherwise.

B.2 Function: AM_AllocateEndpoint

int rval = AM_AllocateEndpoint(eb_t bundle, ep_t *endp)

AM_AllocateEndpoint creates a communication endpoint in the *bundle*. If successful *endp* contains the address of the new endpoint. The function returns AM_OK if the endpoint was added to the bundle and AM_ERR_XXX otherwise.

B.3 Function: AM_Map

int rval = AM_Map(ep_t ea, int index, en_t endpoint, tag_t tag)

AM_Map establishes a translation-table entry mapping such that relative to *ea* the *index* is bound to *endpoint* and *tag*. The *index* must be less than the value returned by *AM_MaxNumTranslation*. An endpoint may map to itself for loop back behavior. It is an error to re-map an active translation-table entry. The function returns AM_OK if the mapping was successfully established and AM_ERR_XXX otherwise.

B.4 Function: AM_MapAny

int rval = AM_MapAny(ep_t ea, int *index, en_t endpoint, tag_t tag)

AM_MapAny establishes a translation-table entry mapping where that the system picks a translation-table entry such that relative to *ea* the *index* is bound to *endpoint* and *tag*. The chosen index is returned in *index*. An endpoint may map itself for loop back behavior. It is an error to re-map an active translation-table entry. The function returns AM_OK if the mapping was successfully established and AM_ERR_XXX otherwise.

B.5 Function: AM_Unmap

int rval = AM_Unmap(ep_t ea, int index)

AM_Unmap removes a previous translation-table entry mapping. The function returns AM_OK if the mapping was successfully removed and AM_ERR_XXX otherwise.

B.6 Function: AM_FreeEndpoint

int rval AM_FreeEndpoint(ep_t ea)

AM_FreeEndpoint atomically deallocates the endpoint. The calling process must have previously created the endpoint. Messages en route to the endpoint may or may not be delivered and/or handled depending on whether or not the endpoint exists when they arrive. The system returns all messages sent to nonexistent endpoints to their senders. The system destroys, without handling, all pending messages in the endpoint's receive pool. The function returns AM_OK if the endpoint was successfully deallocated and AM_ERR_XXX otherwise.

B.7 Function: AM_FreeBundle

int rval = AM_FreeBundle(eb_t bundle)

AM_FreeBundle deallocates the endpoint *bundle*. Any endpoints in the bundle are first removed as if *AM_FreeEndpoint* was called for each one. The function returns AM_OK if the bundle was successfully freed and AM_ERR_XXX otherwise.

B.8 Function: AM_MoveEndpoint

int rval = AM_MoveEndpoint(ep_t ea, eb_t from_bundle, eb_t to_bundle)

AM_MoveEndpoint moves the endpoint *ea* from *from_bundle* to *to_bundle*. This allows movement of endpoints between bundles. The function returns AM_OK if the endpoint was successfully moved and AM_ERR_XXX otherwise.

B.9 Function: AM_SetExpectedResources

int rval = AM_SetExpectedResources(ep_t ea, int n_endpoints, int n_outstanding_requests)

AM_SetExpectedResources requests that the system reserve adequate endpoint communication resources such that the endpoint can efficiently communicate *n_outstanding_requests* with up to *n_endpoints* other endpoints without any messages being returned due to persistent endpoint congestion. The function returns AM_OK if the system could make the reservation of endpoint resources and AM_ERR_XXX otherwise.

B.10 Function: AM_SetTag

int rval = AM_SetTag(ep_t ea, tag_t tag)

AM_SetTag atomically sets the endpoint's tag value to *tag*. After this function returns, messages with tags matching the new tag will be delivered and all others will be returned to their senders. Before this function returns, all pending messages in the endpoint's receive pool are returned to their senders. The function returns AM_OK if the endpoint tag was successfully changed and AM_ERR_XXX otherwise.

B.11 Function: AM_GetTag

int rval = AM_GetTag(ep_t ea, tag_t *tag)

AM_GetTag retrieves the endpoint's tag value int *tag*. The function returns AM_OK if the endpoint tag was successfully retrieved and AM_ERR_XXX otherwise.

B.12 Function: AM_GetTranslationName

int rval = AM_GetTranslationName(ep_t ea, int i, en_t *gan)

AM_GetTranslationName retrieves the global-endpoint name associated with the *ith* entry of the endpoint's translation table and stores it into *gan*. The function returns AM_OK if the endpoint name was stored into *gan* and AM_ERR_XXX otherwise.

B.13 Function: AM_GetTranslationTag

int rval = AM_GetTranslationTag(ep_t ea, int i, tag_t *tag)

AM_GetTranslationTag retrieves the tag associated with the *ith* entry of the endpoint's translation table and stores it into *tag.* The function returns AM_OK if the tag was successfully stored into *tag* and AM_ERR_XXX otherwise.

B.14 Function: AM_GetTranslationInuse

int rval = AM_GetTranslationInuse(ep_t ea, int i)

AM_GetTranslationInuse returns AM_OK if an active translation is installed in the *ith* entry of the endpoint's translation table, and AM_ERR_XXX otherwise.

B.15 Function: AM_MaxNumTranslations

int rval = AM_MaxNumTranslations(int *ntrans)

AM_MaxNumTranslations returns the maximum (zero-based) translation-table index for all endpoints in the system in *ntrans.* This value is at least 256 across all interface implementations.

B.16 Function: AM_GetNumTranslations

int rval = AM_GetNumTranslations(ep_t ea, int *ntrans)

AM_GetNumTranslations returns the number of entries in the endpoint's translation table in *ntrans*. This value is less than the value returned from *AM_MaxNumTranslations.*

B.17 Function: AM_SetNumTranslations

int rval = AM_SetNumTranslations(ep_t ea, int ntrans)

AM_SetNumTranslations sets the number of translation-table entries in the endpoint to *ntrans*. The value of *ntrans* should be at least 256 and less than the value returned from *AM_MaxNumTranslations.* The function returns AM_OK if the number of translation-table entries was successfully set to *n* and AM_ERR_XXX otherwise.

B.18 Function: AM_GetSourceEndpoint

int rval = AM_GetSourceEndpoint(void *token, en_t *gan)

The communication system passes an opaque *token* pointer to each active message handler. *AM_GetSourceEndpoint* takes such pointers and returns the globally unique endpoint name of the endpoint that sent the associated request message into *gan*. The function returns AM_OK if the *token* was successfully translated into a valid global-endpoint name and AM_ERR_XXX otherwise.

B.19 Function: AM_GetDestEndpoint

int rval = AM_GetDestEndpoint(void *token, ep_t *endp)

The communication system passes an opaque *token* pointer to each active message handler. *AM_GetDestEndpoint* takes such tokens and returns the endpoint address of the local receiving endpoint endp. The function returns AM_OK if the *token* was successfully translated into a local-endpoint address and AM_ERR_XXX otherwise.

B.20 Function: AM_GetMsgTag

int rval = AM_GetMsgTag(void *token, tag_t *tagp)

The communication system passes an opaque *token* pointer to each active message handler. *AM_GetMsgTag* takes such tokens and returns the corresponding message tag into *tagp*. The function returns AM_OK if the *token* was successfully translated into a local-endpoint address and AM_ERR_XXX otherwise.

B.21 Function: AM_SetHandler

int rval = AM_SetHandler(ep_t ea, handler_t handler, void (*function)())

AM_SetHandler sets the handler-table entry *handler* for the endpoint *ea*. Once installed, incoming active messages naming *handler* invoke the *function*. The function returns AM_OK if the *handler* function was added and AM_ERR_XXX otherwise.

B.22 Function: AM_SetHandlerAny

int rval = AM_SetHandlerAny(ep_t ea, handler_t *handler, void (*function)())

AM_SetHandlerAny lets the system pick the handler-table entry for the specified handler function for the specified endpoint *ea*, and returns the handler-table index in the *handler* pointer. Once installed, incoming active messages naming *handler* invoke the *function*. The function returns AM_OK if the *handler* function was added and AM_ERR_XXX otherwise.

B.23 Function: AM_GetNumHandlers

int rval = AM_GetNumHandlers(ep_t ea, int *n_handlers)

AM_GetNumHandlers returns the number of handler-table entries for the endpoint in *n_handlers*.

B.24 Function: AM_SetNumHandlers

int rval = AM_SetNumberHandlers(ep_t ea, int n_handlers)

AM_SetNumHandlers sets the number of handler-table entries for the endpoint to *n_handlers*. The number of entries should be at least 256 and smaller than the value returned by *AM_MaxNumHandlers*. The function returns AM_OK if the number of handler-table entries was successfully changed and AM_ERR_XXX otherwise

B.25 Function: AM_MaxNumHandlers

int rval = AM_MaxNumHandlers()

AM_MaxNumHandlers returns the maximum allowable number of handler-table entries for any endpoint in the system. This value is at least 256 across all interface implementations.

B.26 Function: AM_SetSeg

int rval = AM_SetSeg(ea_ t ea, void *addr, int nbytes)

AM_SetSeg sets the endpoint's virtual-memory segment address to *base* and its length to *nbytes.* The base address may have arbitrary alignment though implementations frequently optimize for the double-word aligned (or larger) case. The segment length must be less than the or equal to the value returned by the *AM_MaxSegLength* function. The function returns AM_OK if the endpoint virtual-memory segment base and length were set and AM_ERR_XXX otherwise.

B.27 Function: AM_GetSeg

int rval = AM_GetSeg(ep_t ea, void *addr, int *nbytes)

AM_GetSegBase returns the base address of the endpoint's virtual-memory segment in *addr* and its length in bytes in *nbytes.*

B.28 Function: AM_MaxSegLength

int rval = AM_MaxSegLength(int *nbtyes)

AM_MaxSegLength returns the maximum segment length in bytes supported by the system in *nbytes.*

B.29 Function: AM_GetEventMask

int rval = AM_GetEventMask(eb_t eb)

AM_GetEventMask returns the bundle's event mask value.

B.30 Function: AM_SetEventMask

int rval = AM_SetEventMask(eb_t eb, int mask)

AM_SetEventMask sets the bundle's event mask value. It is closely related to the *AM_WaitSema* function.

This function is atomic in the following sense: if the event condition holds when the mask is being set to enable the condition then the mask is set and the event is *generated.* Whenever an event is *generated,* the system atomically sets the bundle's binary semaphore to one, and clears the corresponding event mask bit. If there are any threads blocked on the semaphore, one is unblocked. For example, a thread may enable an event and wait using *AM_WaitSema* until it occurs. Then is disables the event by calling *AM_SetEventMask* with AM_NOEVENTS, does its work, and resets the mask before calling *AM_WaitSema* again.

The interface defines these masks and events:

- AM_NOEVENTS

  No endpoint-state transition generates an event.
- AM_NOTEMPTY

  A nonempty receive pool or a receive pool that has a message delivered to it generates an event.

B.31 Function: AM_WaitSema

int rval = AM_WaitSema(eb_t eb)

AM_Wait blocks the calling thread until the count in the bundle's semaphore becomes equal to one and then atomically decrements it. When an event occurs the system performs the matching operation, logically AM_PostSvar, which atomically increments the count in the bundle's semaphore and clears the event mask so as to disable subsequent generation of the same event. If there are any threads blocked on the semaphore, one is unblocked.