AMUDP: Active Messages Over UDP

CS294-8 Semester Project, Fall 2000

Dan Bonachea and Dan Hettena

{bonachea,danielh}@cs.berkeley.edu

1 Abstract

Active Messages (AM) is a lightweight messaging protocol used to optimize network communications with an emphasis on reducing latency by removing software overheads associated with buffering and providing applications with direct user-level access to the network hardware [1]. AM provides low-level asymmetric network messaging primitives which have come into popular use as a low-level substrate in the implementations of higher-level parallel languages and systems, for example MPI, Split-C, Titanium, and others [10], [15], [16].

Most implementations of AM are highly hardware-specific, and in particular they usually require the support of high-performance, non-commodity "smart" network interfaces such as Myrinet [12]. This unfortunately presents a problem when trying to run AM-based software on systems that have commodity network interface hardware, or network interfaces for which no AM implementation is readily available. This first part of this project attempts to bridge that gap by providing an AM-2 implementation that runs on UDP, a standard component of the TCP/IP protocol suite that is ubiquitous across platforms and operating systems [13]. We don't expect to achieve latency performance competitive with a native implementation of AM optimized for special purpose hardware, instead we seek to provide a compatibility layer that will allow AM-based systems to quickly get up and running on virtually any platform. The motivation for choosing UDP over other protocols (such as TCP) is that it typically provides the lowest overhead access to the network with little or no internal buffering, and the connectionless model is best suited for scaling up the number of distributed processors. Because UDP occasionally drops packets, we add a thin reliability layer that provides the guaranteed delivery required by AM-2, hopefully providing this fault tolerance with better performance than full-blown TCP.

In the second part of this project, we attempt to further optimize the AM over UDP layer by targeting a specific commodity Ethernet interface (the Intel EtherExpress Pro 100). We provide a special version of the AM layer that bypasses the kernel implementation of UDP for that network interface card (NIC) and uses high-performance direct hardware access with minimal buffering. This provides a high-performance AM implementation on systems that use the Intel EtherExpress Pro 100 commodity hardware (which is very common on x86 architectures), and specifically will provide high-performance active messages on the IStore architecture which also uses that NIC [6].

2 Introduction & Motivation

2.1 AM Applications

Active messages is a lightweight, low-level messaging protocol used in various software systems, and provides high-performance, low-latency network communications with minimal kernel interaction. AM is frequently used as a bottom-level networking layer in parallel programming environments and languages such as MPI and Split-C.

One notable system that currently utilizes active messages as a communications layer is Titanium, a parallel dialect of Java developed at UC Berkeley [16], [19]. Titanium has a global memory space abstraction, allowing applications to concisely address memory on remote processors. As a result, Titanium applications can be quite sensitive to network latencies, so considerable effort is made to use the most lightweight communication protocols available on a given distributed architecture in the implementation of the runtime system.

One of the primary goals of Titanium is portability across a variety of distributed computing platforms, and backends have been written using various lightweight networking layers: active messages, IBM's LAPI protocol, and Cray's SHMEM library. Of these, the most portable and hardware-independent interface is active messages, and AM implementations exist for several specific varieties of network hardware. However, to our knowledge there's no truly portable, hardware-independent implementation of active messages available. AMUDP provides a very portable implementation of AM requiring only UDP, thus providing a quick and easy way for active-message-based systems (such as the Titanium runtime system) to get up and running on virtually any distributed architecture.

2.2 Portability

2.2.1 Why UDP?

The primary reason we chose UDP as our lower-level protocol is that it's the lightest weight network protocol that's widely portable (by virtue of being included in the TCP/IP protocol suite). It provides checksums that prevent corrupted packets from being delivered, but doesn't provide any guarantees about packet delivery or order of arrival that could degrade latency performance.

2.2.2 Why not TCP?

The only other protocol that offers such widespread portability is TCP, so a fair question to ask is, why not TCP? The first reason is latency performance – TCP provides a generic protocol for guaranteed, in-order delivery of stream-based data, and this generality and reliability comes at a price in performance. TCP is also a somewhat complicated protocol, and as a result many implementations incur significant software overheads in data copies and buffer management.

AM guarantees message delivery, but doesn't require in-order delivery. Furthermore, the communication pattern is strictly request-response, and most messages have a small, fixed maximum size. In implementing AMUDP we've taken advantage of these specialized characteristics, augmenting UDP to provide the required delivery guarantees with highly streamlined buffer management and essentially no extraneous network messages.

The second reason why TCP is not a good choice is because it's connection-oriented. Active messages allows messages to flow between arbitrary endpoints within a group of communicating systems, so the number of total TCP connections required to implement active messages over TCP would be quadratic in the number of communicating processors. Because each TCP connection consumes non-trivial resources, such a solution would not scale well to large distributed systems. On the other hand UDP is connectionless and easily allows packets to be sent between arbitrary endpoints in a set of communicating machines, which is an ideal match for the AM model.

2.3 Performance & Fault Tolerance

In addition to portability, we also have the obvious goals of providing performance (in the form of low latency and good bandwidth utilization) and fault tolerance (taking advantage of the redundant IStore NIC hardware, and guaranteeing reliable delivery in the face of hardware packet loss).

3 Design of AMUDP

3.1 Overview

The AMUDP library implements the majority of the Active Messages–2 specification [8], with a few small caveats described in section 3.5. It consists of about 7000 lines of C/C++ code, and took just over a week to complete. The library supports active message applications written in C and C++, and should run on any standard UNIX or Microsoft Windows system that provides UDP (which is included in virtually all TCP/IP implementations). In addition to the functionality required by the AM specification, the AMUDP library also includes a concise interface that provides a portable mechanism for parallel SPMD job startup, and a performance introspection API that monitors library activity and reports on various performance metrics of interest.

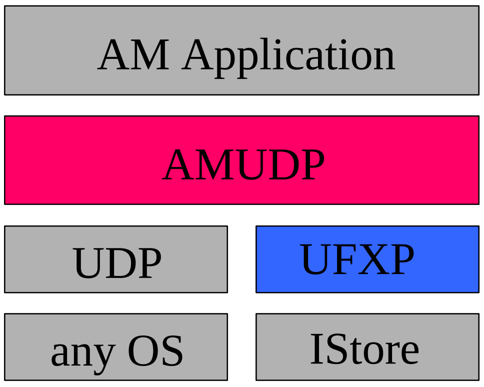


Figure 1: High-level AMUDP System Architecture

Figure 1 demonstrates the high-level system view of an AM-based application running on AMUDP. As shown, AMUDP currently runs on one of two possible backends (selected at library compilation time). The UDP backend, which is very portable and should run on any reasonable OS and hardware, and the optimized, user-level UFXP backend, which is specific to the Intel EtherExpress Pro 100 NIC used in the IStore architecture.

3.2 Portability

3.2.1 AMUDP runs everywhere

Traditionally, implementations of active messages have been very hardware-dependent and platform-dependent, primarily to minimize software latencies and maximize performance. Most accomplish this by exploiting some mechanism for direct user-level access to the network hardware without kernel intervention, which is inherently a hardware-dependent optimization in current operating systems. AMUDP sacrifices a small amount of performance to achieve widespread portability and ease of use. By using the standard UDP driver, we incur the performance penalty of kernel interaction, but gain the ability to run on any platform and even in environments with heterogeneous network hardware and/or operating systems.

3.2.2 Portable, automatic job startup and endpoint naming system

The AM-2 specification intentionally avoids any position on how an application obtains the names and tags of remote endpoints – it merely provides functions for allocating endpoints, retrieving opaque name values from them, and setting opaque names and tags in the translation tables of an endpoint. Furthermore, they provide absolutely no support or specification describing how a parallel AM job gets initiated across the nodes in a distributed computing environment. These design decisions were wise in the interests of creating a portable specification that could be implemented on many distributed architectures, because job startup and naming systems are frequently implementation dependent. However, the unfortunate result was that most AM implementations also provide no direct support for spawning jobs or obtaining remote endpoint names – rather, they generally choose to rely on external utilities which are often platform-dependent and even site-dependent for accomplishing these tasks. This led to lengthy, configuration-dependent and often error-prone “boiler-plate” initialization code appearing in the main() functions of most AM applications, which generally increased the complexity of writing AM-based applications and obscured some of their functionality.

In pursuit of the goal of creating a truly portable AM-2 implementation, the implementation of AMUDP also includes a library that provides portable job startup and endpoint initialization services for SPMD-like AM applications. SPMD-based AM applications can perform virtually all their initialization tasks on AMUDP by calling a single library routine (AMUDP_SPMDStartup()) very early in their main function, and providing the number of processors in the job and a copy of the command-line arguments. The implementation of AMUDP_SPMDStartup() automatically takes care of spawning the correct number of remote processes, establishing connectivity between them (by creating endpoints and setting up their translation tables and tags correctly), and establishes services for forwarding input and output between the worker processes and the user console as necessary. On the worker processes, the call to AMUDP_SPMDStartup() returns with an endpoint that is essentially ready for use – all the application has to do is register its AM handler functions by calling AM_SetHandler, and it can start using the endpoint. In addition, AMUDP provides calls for obtaining processor rank and parallel degree, terminating a parallel job, and executing a nonoptimal, but functional barrier (useful for bootstrapping synchronization-related handlers).

The job spawning mechanism in AMUDP_SPMDStartup() works by executing a callback to a provided function pointer argument with the name and arguments for the remote worker process that must somehow be spawned on the remote nodes. All the spawning function has to accomplish is for a process to be created on the remote node running the current executable with the given arguments, and AMUDP can take it from there. This is the one inseparably platform-dependent component of the initialization process, as many cluster sites provide their users with locally developed scripts or programs used for remote job spawning. The AMUDP library includes default implementations of this spawning function for several common spawning mechanisms, namely: ssh remote shell, rexec, GLUNIX spawn, and local process spawn (for debugging purposes) – their implementations range in size from 4 lines (for local spawn) to about 100 (for ssh spawn), and creating new spawning functions to accommodate different site conventions should be very easy.

Figure 2 illustrates the architecture for the job startup system that shows how the necessary coordination is accomplished. When the master node (the job spawned locally by the user) enters the AMUDP_SPMDStartup() function, it binds a TCP port that is used to bootstrap communication with the worker nodes by passing them the master port address through "hidden" command line arguments. Once the spawning function causes the remote copies of the AMUDP application to start running, they also call the AMUDP_SPMDStartup() function, which uses the hidden arguments to recognize it is a worker process, and uses the master port to connect to the master process and establish communication. Each worker sends the master the name of a newly created endpoint (an IP address/port combination for the UDP backend). Upon hearing from the last worker, the master sends each worker node a rank, a tag, and a copy of the other endpoint's names and tags for insertion in their endpoint translation table (also some other general info such as job parallel degree).

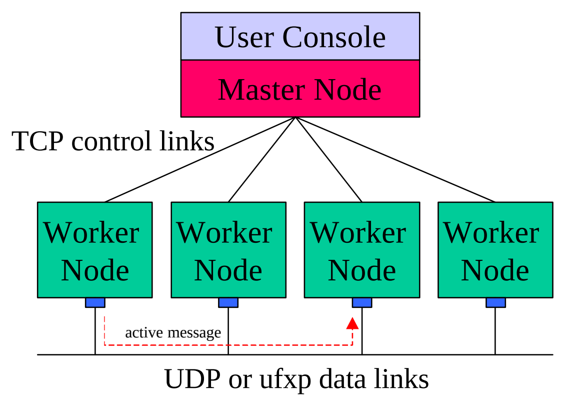


Figure 2: AMUDP Job startup and control architecture

3.3 Overview of the HPAM protocol

The AM-2 specification guarantees that all messages will be delivered, or returned to their sender with an explanatory error code. Because UDP is an unreliable protocol, one of the primary challenges in implementing AM-2 over UDP was adding a low-cost reliability mechanism.

The design of our reliability protocol was heavily influenced by the HPAM protocol [9], which was used to implement a restricted form of active messages over an unreliable datagram protocol resembling UDP. The protocol handles almost every situation we care about, and aggressively takes advantage of AM's strict request-reply model to provide streamlined buffer management and reliable delivery with zero overhead in the common case of no packet loss due to hardware failure.

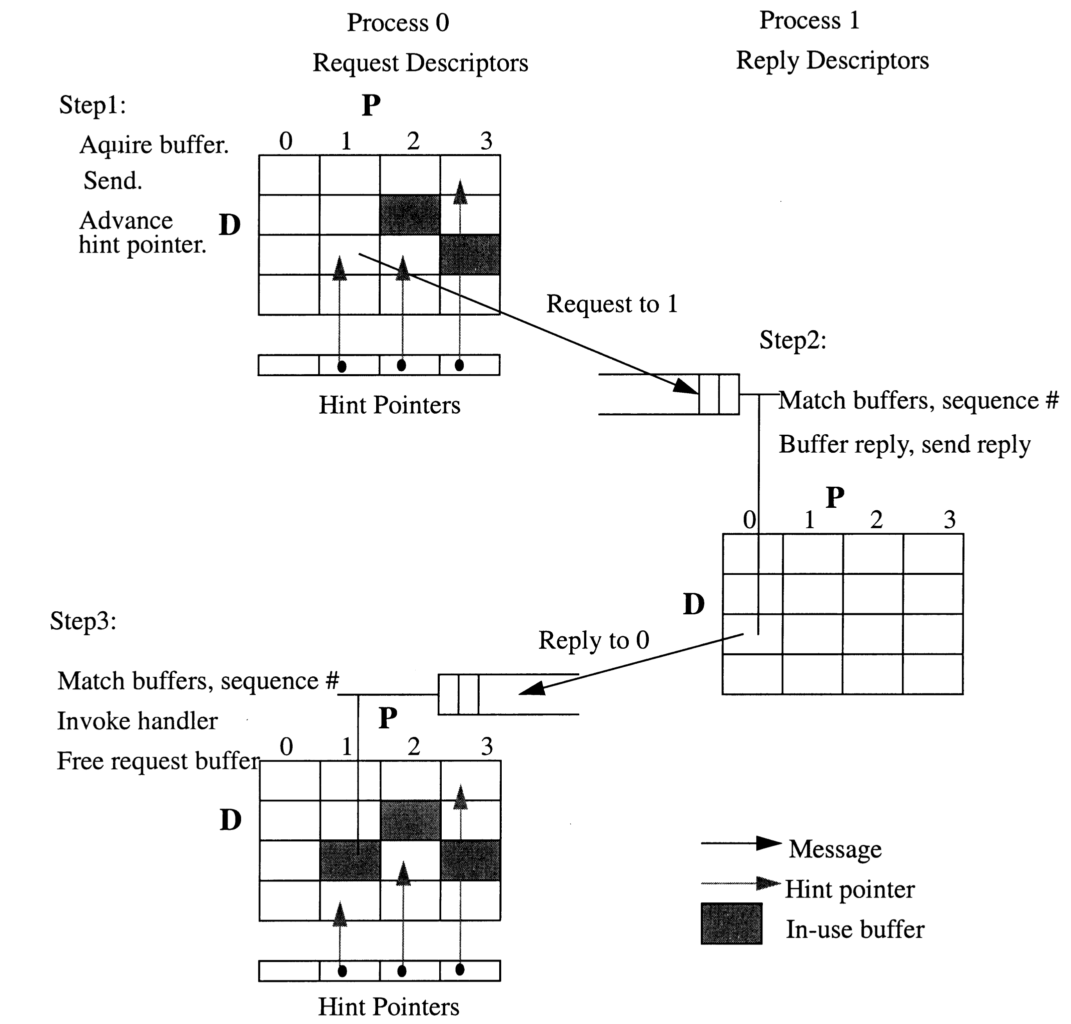


FIGURE 4. Connection state maintained by the all-to-all D-way protocol

Figure 3: The HPAM reliability protocol (D=4)

Figure 3 (borrowed from [9]) illustrates the basic operation of the protocol. At the core of the protocol is the idea that because AM is strictly request-reply, it's possible to perform allocation for all the network buffers needed to send and receive a request and its corresponding reply along the entire network path before sending the request. At the start of the parallel job, we set a fixed network depth (D) which is the maximum number of outstanding requests allowed in the system at any time (requests which haven't yet received a reply). Each processor allocates 2*(P-1)*D send buffers and 2*(P-1)*D receive buffers (where P is the parallel degree), which ensures that once a request is sent, there is always a free buffer available along the network path the message and its corresponding reply must travel, so we never deadlock or drop packets due to congestion. Thus, the buffer allocation problem reduces to selecting a free “instance” (where there are D instances for each processor-processor pair) of the protocol at the request initiator which corresponds to a free request receive buffer on the replying node, a specific reply send buffer at the replying node, and a reply receive buffer on the requesting node. The “hint” pointers pictured above are used to optimize this selection, which are managed such that in the common case they always point to a free instance.

Reliability is achieved by leveraging the reply message as an ACK that indicates the corresponding instance can be freed, so in the common case we send no extra messages. The requestor uses a timestamp to decide when the request or corresponding reply should be considered lost and retransmits the request from the same send buffer. If the reply is the one that got lost (i.e. after the request handler ran), the replying node detects this condition using a 1-bit sequence number and resends the reply from the instance reply buffer (this enforces the AM invariant that handlers run exactly once or not at all).

The interested reader is encouraged to consult [9] for further details on the protocol’s operation.

3.4 Fault tolerance

3.4.1 Guaranteed delivery and returned messages

AM requires that all messages accepted for delivery are either successfully delivered to their destinations (and cause the corresponding handler to run exactly once) or are returned to the sender where they execute a “returned message” handler with an error code stating the reason why the message was rejected by the remote node (including one for persistant network congestion at the destination or catastrophic network failure). These semantics are fully implemented by AMUDP and support the construction of more robust active-message based applications.

Guaranteed delivery is provided through request retransmissions when a corresponding reply isn’t received within a given timeout window, which is grown using exponential back-off with each loss (on a per-packet basis) to prevent flooding the destination. Incidentally, retransmissions are the only case in the HPAM protocol that can result in messages to be dropped on the receiver side due to insufficient buffer space, so we set the timeout high enough to avoid re-transmitting a packet unless we can be reasonable certain that it was really dropped. Empirical evidence has shown that non-faulty, modern, wired networks are extremely unlikely to ever drop packets for any reason other than congestion, so we feel justified in optimizing protocol performance for the common case of no packet loss (provided we still ensure correct behavior in the presence of packet loss).

Given guaranteed delivery, returned messages are implemented fairly easily by performing the series of required tests (tag check, length check, etc) on all received packets and transmitting them back to the sender with a return-to-sender error code as necessary. Requests that exceed a maximum timeout (currently 30 seconds) are presumed undeliverable due to persistent congestion at the destination or catastrophic network failure and are returned to the application handler by the requesting node.

3.4.2 Fault injection capabilities

In order to properly evaluate the functionality and performance of AMUDP in the face of high transient network failure rates (such as one may find with a bad network cable) we added a fault-injection feature to AMUDP that directs it to drop a fraction of all received messages as if they'd never arrived. This fault injection experiment revealed a heisenbug in the implementation of the request retransmission algorithm that would have been very difficult to find any other way. Once that problem was fixed, we were able to verify that AMUDP continues to operate correctly even with high packet loss rates (results are shown in section 5).

3.4.3 NIC fail-over transparent to the AM application

A final fault tolerance feature of AMUDP is that the UFXP backend of AMUDP performs NIC fail-over on systems equipped with multiple network cards (such as IStore) in a way that should be totally transparent to the application (providing it's running under the SPMD job startup API). When a NIC failure notification is posted to the AMUDP library by the UFXP driver, the information about the old and new endpoint address is sent to the master node, which then propagates the information out to all the worker nodes. The worker nodes silently update the contents of all their endpoint translation tables to reflect the address change, thereby routing new requests and retransmissions for dropped requests (possibly lost with the card) to the new NIC.

3.5 How AMUDP differs from the AM-2 Specification

AMUDP implements the vast majority of the AM-2 specification [8], with a few small exceptions as detailed in the sections that follow. For the most part, these deviations were perceived necessary to satisfy our performance and portability goals, and we feel they shouldn't pose any problems for the AM-based applications of interest.

3.5.1 General Limitations

- We don't implement concurrent bundle/endpoint access (AM_PAR type) – The AM-2 spec allows users to create a bundle with a parallel access type, which means the library performs the locking necessary to allow multiple application threads to concurrently call AM entry points. We currently only implement the sequential access type, primarily to avoid the platform dependent nature of thread locking primitives – applications can trivially ensure sequential access by placing locks around all calls to AM functions (Titanium already does this to handle other AM implementations with similar restrictions). However, we're considering adding support for AM_PAR in the future, because library-provided locking should achieve far greater concurrency than application-level locking, providing better performance for heavily multi-threaded AM applications.

- An implicit assumption in the HPAM protocol is that all endpoints are told at job startup time the names of all the other endpoints with which they communicate directly (send messages to or receive messages from). As a result, our AMUDP library is slightly less general than the AM-2 spec, requiring application behavior somewhat closer to the typical SPMD model. AM applications running on AMUDP *must* call AM_SetExpectedResources() exactly once on an endpoint after setting up the translation table and before making any calls to the transport functions. It is an error to call AM_Poll, AM_Reply* or AM_Request* before the call to AM_SetExpectedResources(). It is also an error to call AM_Map, AM_MapAny or AM_Unmap (which change the translation table) after the call to AM_SetExpectedResources(). This has the effect of "freezing" the endpoint's translation table to the list of endpoints to which it may send active messages. Similarly, active messages are only accepted from endpoints listed in the translation table of the destination endpoint - all other messages are returned to sender. Note that we still permit modification of remote endpoint tags on the fly (which normally is handled by re-mapping the remote endpoint using AM_Map) - the new entry point AMUDP_SetTranslationTag() allows this to occur after the call to AM_SetExpectedResources(). Regarding the specific call to AM_SetExpectedResources(), the n_endpoints parameter is ignored, but the n_outstanding_requests is taken to be the network depth parameter for the HPAM protocol, and therefore has a direct impact on the memory usage and performance of the network layer - setting it too large may cause an error to be returned.
- Finally, we don't support taking the address of AM_ entry point "functions" (because many are implemented as macros)

3.5.2 Limitations when running AMUDP on the UFXP backend

- We don't allow more than one concurrent endpoint per application while running on UFXP, because we currently bind each AM endpoint directly to a hardware NIC. In the future, we may provide a way to multiplex multiple AM endpoints on a single UFXP NIC.
- When using UFXP, en_t names of remote and local endpoints may change during any AM_ transport call (to provide fault-tolerance in the face of remote NIC failures), so the application should not keep copies of these around in local data structures across AM transport calls. This should not be a serious limitation because en_t objects are opaque to the AM application, so the only thing it can really do with them is retrieve them from remote nodes and place them in the endpoint translation table (where they will automatically be fixed up upon remote failure).

3.5.3 Restrictions

3.5.3.1 Job startup API

The job startup API only works for AM applications that can be cast within a SPMD-like communication model (where all processors can communicate with all others), as this was perceived to be the most common type of AM application, and probably general enough to encompass most models that others that users may wish to use. Note the AMUDP implementation of the AM entry points doesn’t require SPMD behavior (beyond the limitations described above) – it also permits more bizarre communication patterns allowable under the AM spec (for example, a pattern where endpoints are restricted to communicate only with nearest neighbors in some N-dimensional grid), it simply doesn’t provide automatic job creation functionality for such non-SPMD configurations.

3.5.3.2 Bulk Transfers

The AM-2 specification provides several functions for supporting bulk requests and replies, and requires that all implementations support these bulk transfers with a maximum transfer size of at least 8192 bytes. Because it’s necessary to support retransmitting any lost packets (including those which may be part of a bulk transfer) at a later time, we must keep a copy of the transferred data somewhere in the communication layer. The most natural solution would be to set our buffer size large enough to handle the largest bulk transfer (that is, use 8192-byte buffers) – however, the 8KB minimum that seems reasonable at first glance quickly proves non-workable and wasteful for this design (for example, in a 64-processor job with a network depth of 10, this design would require 20 MB of memory for the communication buffers alone, most of which would probably never be used).

The original HPAM protocol makes no provision for implementing “bulk” transfers, that is, where the size of the requested bulk transfer exceeds the size of a single HPAM buffer (in their case, 4500 bytes). The paper claims this functionality can be built on top of the provided medium-sized messages – while this may be true from within an application that has knowledge of its memory behavior, we contend it’s not possible to provide the semantics required by the AM bulk transfer functions on top of the HPAM protocol without violating the strict request-reply model of AM or imposing significant dynamic buffer management and data copying. Specifically, the AM function AM_ReplyXferM can be called from any request handler and directs the communication system to send a bulk reply to the requesting node. However, the HPAM protocol performs all buffer allocation on the requesting node, so if the bulk reply size exceeds the capacity of one HPAM buffer, there’s no way for the replying node to allocate the necessary additional send buffers on the reply node and additional receive buffers on requesting node. The requesting node has no way to anticipate a priori whether the replying node will need additional buffers, because AM_ReplyXferM can be called from any request handler.

Complicated schemes involving a roundtrip to the requesting node within the AM_ReplyXferM function violate the non-blocking semantics of the reply handler that is running, and could potentially deadlock the system because they violate the strict request-reply model and introduce a new buffer dependency.

While bulk transfer replies are outside the scope of what the HPAM protocol can handle, it is possible to implement bulk transfer requests, where the requesting node initiates a bulk transfer to or from the replying node. We have implemented the AM entry points (AM_RequestXferM and AM_RequestXferAsyncM) that provide synchronous and asynchronous bulk transfer “push” operations. They accept bulk transfers up to 128KB in size, and perform fragmentation of the transfer buffer into as many 512-byte HPAM buffers as necessary. The fragments travel independently and in parallel to the replying node (any retransmissions necessary only retransmit the lost fragments), where they are reassembled by the replying node into the endpoint’s memory segment at the offset indicated by the argument to the request function. The arrival of each non-final fragment does not trigger a user-level handler, but does generate a silent reply packet that informs the requestor each fragment arrived and the instance can be freed. As required by the AM spec, the request handler is run on the entire destination buffer once the final fragment has arrived and been copied into place. In order to make this strategy work, each fragment carries a count of the number of fragments involved in its bulk transfer, and a bulk transfer sequence number which is used to disambiguate fragments from independent concurrent bulk transfers so the replying node can properly update its view on the progress of each transfer. Large transfers that require more fragments than the number of free instances will block in a polling loop as fragments are sent across the network until the last fragment gets placed into a send buffer. The AM spec also includes a request function (AM_GetXferM) that performs a bulk transfer “pull” operation, which can be implemented in a similar fashion (the prototype doesn’t currently implement AM_GetXfer due to time constraints).

4 Design of UFXP

The UFXP layer is targeted at ISTORE [3],[6], which will be a cluster of small PC-like bricks used to study highly available, maintainable, and adaptive storage-intensive network services. Each ISTORE brick has hardware devoted to fault-tolerance and fault-generation, including multiple Intel EtherExpress Pro 100 network interfaces.

These network interfaces are commodity parts, and therefore particularly inexpensive. Each network interface can operate at 100Mbps, and each brick has a disk that can be accessed at almost 40MBps, so each brick receives four EtherExpress Pro 100 (hereon abbreviated FXP) network interfaces.

Normally, a brick will use all four of its network interfaces in parallel, using a transparent fault-tolerant IP striping algorithm. This is ideal for TCP- and UDP-based applications, which will automatically have access to 400Mbps of network bandwidth transparently routed across four links. Their latency will also be improved, since there will be four times fewer packets queued on each link. Further, if part of the network fails, the IP striping system will automatically stop using the affected network interfaces. AMUDP, which can run on top of UDP, will be able to take advantage of these features automatically.

However, some applications are very latency-sensitive. Network protocol stacks are traditionally implemented as part of an operating system kernel, but this adversely affects the latencies of application network send and receive operations, because interactions between applications and the kernel are typically slow.

Therefore a latency-sensitive application can achieve significantly greater performance if interaction with the kernel is removed from the critical paths of sending and receiving packets. However, in order to send and receive packets, the processor must directly access the network interface registers. For security reasons, normally only the kernel can access device registers; otherwise a malicious application could use a hardware device to read and write arbitrary bytes of main memory.

Some network devices, such as Myrinet interfaces, are intelligent enough to prevent such application behavior. Unfortunately this intelligence is extremely costly, and commodity network devices such as the FXP do not implement it.

But if the application is trusted, the kernel can give the application just enough permission so that, while the kernel will be removed from the critical path of sending and receiving packets, it is also unlikely that the application will accidentally harm the system. This is the goal of the UFXP layer.

The UFXP layer does not attempt to forward IP packets to the kernel; this would be very inefficient. Instead, when the UFXP layer claims an Ethernet interface, the kernel IP striping system stops using the interface. Because each ISTORE brick has four Ethernet interfaces, the kernel can easily tolerate sacrificing one for a latency-sensitive application.

The UFXP layer is fault-tolerant as well. When it detects that its Ethernet interface is no longer functioning properly, it is able to cleanly flip to another of the four interfaces. The UFXP layer sends an interface address change notification to the AMUDP layer, which in turn is responsible for making the change transparent to the AM user.

The UFXP layer, approximately 3000 lines of code, is composed of a modified Linux kernel driver and a user-level library. The kernel driver is based upon the original Linux driver for the FXP. Much of the user library is actually derived from the kernel driver.

Most of the driver is unmodified, as it was only necessary to make a few additions. Specifically, the driver maps FXP's register set into the UFXP application's virtual address space. The driver also forwards network interrupts to the library via UNIX signals. UNIX system calls do not allow non-root processes to pin memory, so the driver provides this functionality as well. Finally, the driver must make sure that the kernel always owns at least one functioning Ethernet interface.

To prevent malicious applications from directly accessing the FXP, the kernel driver is accessible to user processes only through a special file in /dev, and so it is possible to restrict access to it using UNIX file permissions.

One of the responsibilities of the user-level library is to use the memory-mapped device registers to manage the transmission and reception of packets to/from the network. The code for manipulating these registers is, of course, not at all portable. However, this is actually the simpler portion of the library. One of the library's more complicated responsibilities is to manage packet buffers in the pinned memory space. Because of inherent similarities in how PCI Ethernet chips manage packet buffers, this code is actually fairly portable.

Consider first a standard non-PCI network card. To send a packet, a driver copies the packet data from main memory to the NIC's RAM, and then sends a transmit command to the NIC. When a packet arrives, the NIC issues an interrupt, and the driver's interrupt handler copies the packet data from the NIC's RAM to main memory. In both cases, the driver communicates with the NIC even up to the point where the NIC has a copy of the packet in its RAM. This is inefficient, because it forces the processor to be directly involved in copying between main memory and the NIC's memory. Also, the NIC must interrupt the processor each time the NIC empties its transmit queue or fills its receive queue, and unless the NIC has a lot of memory, this happens often.

PCI network cards exist to eliminate these inefficiencies, which become particularly costly at speeds of 100Mbps and above, at which many thousands of full-length packets can arrive in a second. When a driver needs to send a packet, it simply directs the NIC to the location of the packet in main memory, and sends a transmit command. The NIC then begins reading the packet from main memory through the PCI bus to its RAM. The NIC can even start transmitting the packet on the network before it has read the whole packet from main memory. Similarly, to handle receives efficiently, the NIC must know *beforehand* where it can store a packet in main memory, so that the driver does not need to guide the transfer.

To eliminate unnecessary interrupts, a PCI NIC should also allow packets to be queued in main memory. For example, packets queued to be transmitted are typically arranged in a linked list in main memory, and the NIC need not interrupt until it reaches the end of the linked list.

Other features, such as being able to gather fragmented packets, are not crucial for performance and therefore are not implemented in every PCI NIC. However, most PCI NIC's share the important buffering features. Ultimately this means that much of the UFXP library is portable to other PCI Ethernet interfaces.

Regardless of the particular NIC, the root of the advantage of user-level networking, such as that provided by UFXP, is that the kernel is not involved in the critical path of sending and receiving packets. Normally bytes must be copied between user buffers and kernel buffers and vice versa, but with UFXP, this is not necessary. For example, when running on UFXP, AMUDP composes outgoing messages directly into the message's final transmission buffer.

Active messages are particularly able to take advantage of user-level networking, because active message handlers operate directly on receive buffers, without first copying the contents of the receive buffer to another user buffer. This means that zero-copy networking can be achieved.

5 Performance results

One of the primary goals of active messages is to provide good network messaging performance. Although AMUDP sacrifices some latency for portability, we still strive for the best performance possible. We use a simple and fast static buffer allocation algorithm, combined with the reliability protocol that is optimized for the common case of no packet loss. Our static network depth helps to ensure that packets never get lost as a result of congestion on the receiver.

The following section presents our initial performance results on AMUDP. The library is still somewhat untuned, and we already know of several performance optimizations that we'd like to make but were unable due to time constraints. For example, presetting some of the unchanging members of the send buffers, ensuring favorable L2 cache alignment of the packet buffers, cutting out some extra bytes in the packet headers, and other general performance profiling and tuning.

5.1 Active Message size breakdown

description	size (bytes)
UFXP header	24
AM header	20
AM arguments	0...32
AM data	0…512
Total	42…586

Active Message size breakdown on UFXP

description	size (bytes)
IP header	20
UDP header	8
AM header	20
AM arguments	0...32
AM data	0…512
Total	48…592

Active Message size breakdown on UDP

The tables above provide a breakdown of the network packet utilization when running AMUDP on the UFXP and UDP backends.

The minimum Ethernet packet size is 60 bytes, so ideally we could lower our minimum packet size to under that size (since hardware latencies don't get any lower past that point, although the cost for packet checksum computation still decreases near linearly with size). The UFXP header can be optimized further to remove 8 bytes of overhead (we plan to do this in the near future).

5.2 Test Methodology

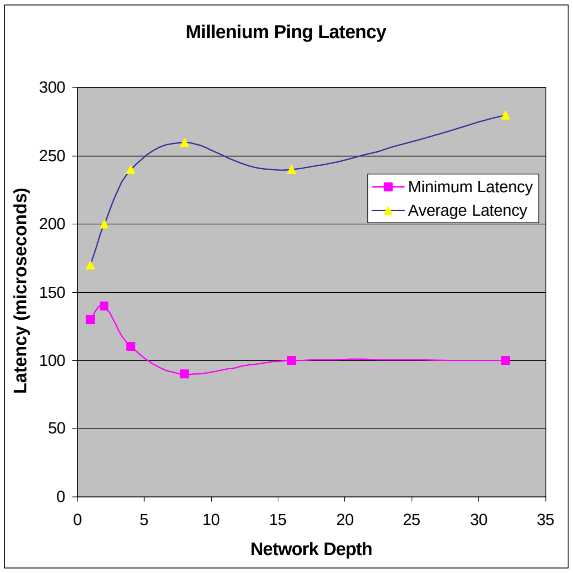
We benchmarked our system using four basic performance tests, described here.

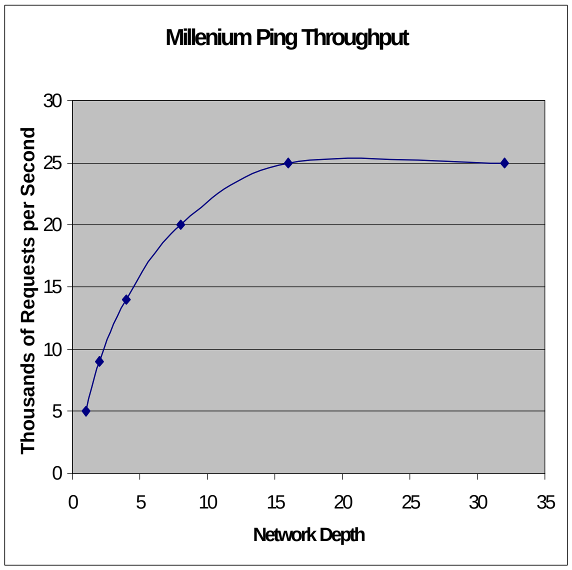
The latency test is a simple 2-node ping application that sends small requests from one node while the other polls. It uses the performance introspection API to measure the latency between the send of each request message and the receive of the corresponding reply message (which includes the time to run a trivial request handler on the replying processor). We report the minimum and average latency times for long runs of ping (10000), using varying network depth (since this parameter has an important impact on performance). We expect the average latency to increase with larger network depths because the receiving processor is kept busier.

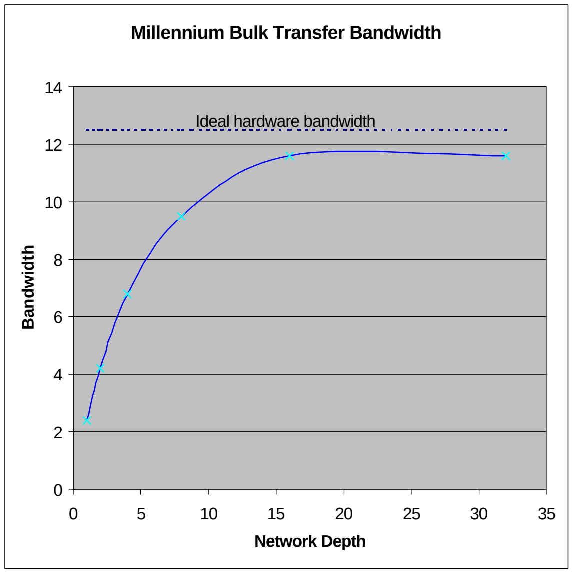
The throughput test uses the same ping application runs, but reports the rate at which requests are injected into the network. We expect the throughput to increase as network depth increases (keeping the network and receiver exceedingly busier), and eventually reach a saturation point where the hardware or software prevents further increase.

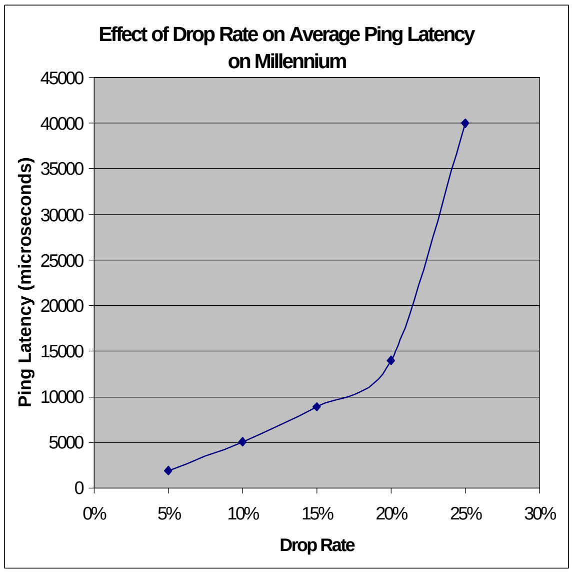
The bandwidth test measures the bulk transfer performance by sending large bulk transfer request messages (8 messages, each 128 KB in length for a total of 1 MB) from one processor to another and measuring the time required to perform the entire transfer at different depths to calculate the bandwidth.

Finally, the fault injection test measures the average latency of the small-message ping application with increasing percentages of dropped packets to assess the performance robustness in the face of transient faults.









5.3 UDP backend on Millennium

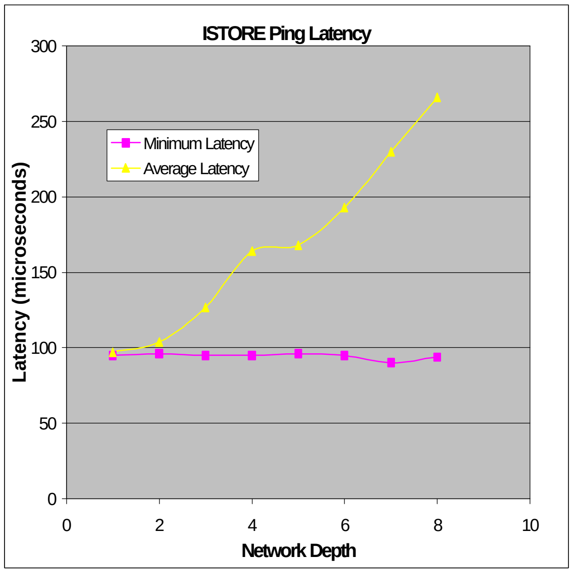
The Berkeley Millennium is a cluster of high-performance dual processor and quad processor x86 SMP’s running Linux, and connected with a 100 MBit half-duplex Ethernet LAN (they also share a high-performance Myrinet network not used in these measurements – the kernel UDP driver uses the Ethernet network).

The charts above demonstrate the latency, throughput performance, bulk transfer bandwidth and latency in the presence of faults achieved on the Millennium cluster using AMUDP.

5.4 UFXP on IStore

The following graph shows the latency performance of the UFXP layer on IStore.

As of this writing, we still had an undiagnosed heisenbug in the UFXP driver causing packet loss rates ranging from 1%-8%, which made it impossible to get accurate throughput, bandwidth or fault injection numbers on that backend. The latency is probably somewhat adversely affected as well.



6 Related Work

6.1 U-Net

The U-Net project [2], [4] was also concerned with fast user-level networking. U-Net virtualizes a network interface to multiple untrusted applications. U-Net supports the DEC DC21140 100Mbps Ethernet controller [18], which is very similar to the Intel EtherExpress Pro 100. As opposed to UFXP, U-Net sacrifices some of its performance in order to achieve protection.

6.2 HPAM

HPAM is an implementation of active messages–1 (AM-1) for a non-commodity, high-performance NIC (the HP Medusa FDDI network card). Their underlying network transport has similar delivery guarantees as UDP, so it was necessary for them to add reliability in software as in AMUDP. They also implemented user-level access to the network card, although they implemented a more complicated scheme that allows a single network card to be multiplexed amongst concurrent trusted applications and the kernel. The multiplexing works by using a special driver that runs as context-switch time that saves/restores NIC state and routes packets accordingly that arrived while the wrong application was running. We admired this design decision, but decided it was unnecessary and possibly detrimental to performance on IStore where multiple hardware NIC's are readily available. Other notable differences between HPAM and AMUDP is that HPAM only implements AM-1 (leaving out much of the functionality in the AM-2 compliant implementation of AMUDP) and they provide no support for bulk transfers.

6.3 PVM-AM

PVM-AM [14] is an implementation of PVM that uses an active-message-like signaling style to implement the communication layer that provides PVM functionality to a user application. Notably, they also used the UDP protocol in the interests of portability, however their implementation relies strongly on the use of UNIX signals to provide message arrival notification, which seems to degrade their performance. Their "active messages" are also significantly different from those in the AM-1/AM-2 specifications, as the only "handler" which apparently ever runs consists of a concurrent library thread that runs asynchronously with respect to the computation thread and buffers the message in user memory where it remains until the user explicitly executes a blocking recv call.

6.4 Active Messages

The original active message specification "AM-1" [5], was released in an initial form in 1992, and had several restrictions on use that specifically targeted SPMD applications. The updated "AM-2" specification [8], was released in 1996, and added support for more general parallel computation models, modularity, multi-threading, and concurrent endpoints and bundles.

6.4.1 Comments on the AM-2 specification

In implementing the AM-2 spec, several minor ambiguities and omissions were discovered. Beyond simple typos in the API specification, there were some notable issues which arose and we feel are useful to document for future developers of AM libraries and applications. As far as we know, this is the first fully-independent implementation of AM-2.

- AM functions return the constant AM_OK on success or one of several error constants on error. However, the constants defined by the API don't provide enough error codes to capture all the possible error situations that one may wish to disambiguate, and the actual semantics of which functions may return which errors and what they mean is entirely unspecified (every function description has a single sentence that ends with "returns AM_OK if successful and AM_ERR_XXX otherwise."). We feel this is a serious omission that single-handedly makes it nearly impossible to write robust, fault-tolerant applications for active messages, and makes it less likely that AM applications will work without change on different AM implementations.
- AM_Startup/AM_Terminate – the semantics of multiple calls is unclear (do they nest?), especially with respect to which bundle and endpoint resources get released when. This is especially important for applications with multiple, independent AM-enabled software components (a design which is specifically encouraged by AM-2).

AMUDP solves this by keeping a counter of the number of open AM sessions (incremented on successful AM_Startup, decremented on AM_Terminate) and releasing all resources when the counter reaches zero, however this solution is imperfect and may create unexpected results if some software component relies on the AM_Terminate call to free its resources.

- It's unclear where the implementation of AM_ReplyXXX should get the tag for the requesting processor to use in the reply, since the requesting endpoint isn't necessarily in the translation table. It seems like a violation of the admittedly weak security mechanism of tags to include the requestor's tag in the request (AMUDP refuses requests whose endpoint have no entry in the translation table, making this issue moot).
- It's also unclear whether tag checks should take place on returned messages, and if so where the returning processor gets the correct tag (AMUDP doesn't check tags on returned messages in order to avoid leaking the send buffer and prevent a returned-message-bad-tag message from cycling endlessly in the network).
- The asynchronous entry points (AM_RequestXferAsyncM and AM_GetXferM) are well intentioned, but poorly thought out. There are three important possible error cases that can arise which in general require the application to reach in different ways: the requested message is too large for the communication subsystem to handle without blocking right now, the requested message is too large for the communication subsystem to *ever* handle without blocking, and some other failure has occurred. Because the spec omits the semantics of any error returns, these three cases are indistinguishable to the application, making these entry points essentially useless for robust applications. Finally, it's unclear whether the asynchronous entry points are allowed to poll (thereby potentially freeing some resources) because this may not constitute returning "immediately" if some handlers block.
- The specification for the returned message handler indicates the AM library passes a "structure" to the handler containing the original message arguments and message token, but gives no details on the layout of this structure or how the application can inspect it (making it essentially worthless to the application developer).
- AM_GetTranslationInuse returns the opposite of what one might expect - AM_OK (zero) for in use, and AM_ERR_XXX (non-zero) for not in use.

7 Future Work

7.1 Titanium

One of our motivations for developing this implementation of active messages over UDP was to provide a highly portable backend for the Titanium runtime system. In the near future we hope to modify the active message backend of Titanium so it can be configured to use AMUDP on platforms where no hardware-specific active message implementation is available, thereby enabling Titanium applications to run on virtually any distributed platform. We are also eager to see the performance of Titanium-based applications on IStore using the UFXP driver.

7.2 NIC striping with user-level access

One possible future direction for the UFXP layer is to support the concurrent use of more than one user-level NIC in a single application, for the purpose of reducing average latency and increasing bandwidth by striping active messages across the network interfaces. We considered implementing this feature, but were unable to do so due to time constraints.

7.3 Clusters of clusters of nodes

One current research area is how to properly support multi-level parallel networking environments, such as those which arise in clusters of symmetric multi-processors (Clumps). In [7], the authors investigate ways to optimize Clumps that use AM between boxes. We feel some of these ideas can be extended further to include clusters of Clumps. For example, the Berkeley Millennium cluster consists of Myrinet-connected Clumps sitting on a gigabit ethernet WAN with other similar Clumps. In order to run an active message application across CLUMPS in such a configuration and make effective use of the available hardware, one would need to send the active messages using different underlying protocols based on the location of the source and destination. For example, in this case an AM application could use a Myrinet-specific active message library for messages to endpoints within its local cluster, and AMUDP for messages with destinations in different clusters across the WAN. It may even be possible to provide an AM "wrapper" library that provides multi-protocol AM delivery transparently to the application (by maintaining an AM2-Myrinet endpoint and an AMUDP endpoint for each wrapper endpoint and multiplexing/demultiplexing accordingly). The only potential problems we see with such a scheme are AM library entry point naming conflicts between the three libraries (which could probably be solved with some clever hackery), and some thought would have to be given to the endpoint naming system and how to infer the correct protocol to use based on the destination name.

7.4 Endpoint migration across nodes

Another area of future work is to provide a mechanism for the migration of endpoints across nodes to tolerate node failure in fault-tolerant AM applications. The AM specification provides enough data abstraction with respect to endpoint names to allow transparent endpoint renaming, which is how we were able to implement transparent NIC fail-over in UFXP. The same mechanism could be used to allow endpoint migration to new nodes. For example, after a failure the replacement node could make a call telling AM that it's taking over all messages previously destined for a given, failed node – messages destined for the failed node would then get automatically routed to the new node with no application-level help from the other nodes.

8 Conclusions

AMUDP provides a highly portable and complete implementation of the AM-2 specification over UDP that should allow active message applications to get up and running on new distributed hardware in no time. In addition AMUDP provides useful API's for platform-independent parallel SPMD job creation and low-level performance introspection. The UDP backend achieves excellent bandwidth utilization and decent latency performance on the Berkeley Millennium cluster, despite the fact that it's using a kernel-level UDP implementation and a commodity 100 Mbit Ethernet.

The alternate UFXP backend for AMUDP provides fully user-level, zero-copy/one-copy active messaging for the Intel EtherExpress Pro 100 commodity NIC on Linux, with a resulting improvement in latency performance. In addition, the UFXP layer enabled AMUDP to provide totally transparent fault-tolerance to applications in the face of NIC hardware failure.

9 References

[1] Active Messages home page. http://now.cs.berkeley.edu/AM/active_messages.html

[2] Basu A., Buch V., Vogels W., von Eicken T.. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado, December 3-6, 1995.

[3] Brown, A., D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D.A. Patterson. "ISTORE: Introspective Storage for Data-Intensive Network Services." Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII), Rio Rico, Arizona, March 1999.

[4] von Eicken, T. Basu, A. Buch, V. and Vogels, W. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing." Proc. Of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado, December 1995.

[5] von Eicken, T. Culler D. Goldstein S. and Schauser K. "Active Messages: a Mechanism for Integrated Communication and Computation", UC Berkeley Technical Report UCB/CSD 92/#675, March 1992.

[6] ISTORE home page. http://iram.cs.berkeley.edu/istore/

[7] Lumetta, S. Mainwaring, A. and Culler, D.; "Multi-Protocol Active Messages on a Cluster of SMP's", Proceedings of Super Computing 1997.

[8] Mainwaring A. and Culler, D. "Active Message Applications Programming Interface and Communication Subsystem Organization", UC Berkeley Draft Technical Report, September, 1996.

[9] Martin, Richard. "HPAM: An Active Message Layer for a Network of HP Workstations", Hot Interconnects II, 1997.

[10] Message Passing Interface (MPI) Forum home page. http://www.mpi-forum.org/

[11] Millennium home page. http://www.millennium.berkeley.edu/

[12] Myrinet home page. http://www.myricom.com/

[13] RFC 758, UDP Specification. http://www.ietf.org/rfc/rfc0768.txt

[14] Subramaniam, K. Kothari, S. and Heller, D. "A Communication Library Using Active Messages to Improve Performance of PVM"Iowa State University, June 1997.

[15] Split-C home page. http://www.cs.berkeley.edu/Research/Projects/parallel/castle/split-c/

[16] Titanium home page. http://www.cs.berkeley.edu/Research/Projects/titanium/

[17] Virtual Interface Architecture (VIA) Specification, Version 1.0. December 16, 1997.

[18] Welsh, M. Basu, A. and von Eicken, T. "Low-Latency Communication over Fast Ethernet", , EuroPar '96, Lyon, France, August 1996.

[19] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, Aiken, “Titanium: A High-Performance Java Dialect”, ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford, California, February 1998.