Cornell University
Dept. of Computer Science
CS 614 – Adv. Systems, Spring '99

Active Containers: A Toolkit for Distributed Shared Memory

Ted Bonkenburg and Vicky Weissman
{tb, vickyw}@cs.cornell.edu

Abstract

Numerous protocols have been designed to implement memory coherency on clusters. The majority of these approaches support multiple readers and multiple writers accessing shared memory. By weakening this constraint we have designed and implemented a high performance, relatively transparent toolkit for multiple reader, single writer shared objects.

The toolkit achieves a high degree of performance by using our implementation of Active Messages 2 to update copies of shared objects. Other key aspects of our design include function shipping and a lazy evaluation of updates.

1 Introduction

In recent years, clusters of inexpensive computers connected by high-speed user level networks have become a viable alternative to shared memory multiprocessor machines. The increasing use of clusters has lead to a growing need for efficient memory coherency protocols in distributed systems. Although a complete shared memory system may be required to port existing parallel computing applications to a cluster, several applications only need a subset of this functionality. Our paper relies on the idea that it is useful to have a single writer, multiple reader shared memory solution that is easy to use. Such a solution can be implemented as a toolkit. We have created a toolkit as a C++ library that can be seamlessly incorporated into an application. We call it *Active Containers*. The toolkit relies on a low-latency communication layer that we built according to the Active Message 2 (AM2) API. The Active Containers toolkit is useful for implementing replication schemes such as process pairs, applications that require shared state such as a cluster wide task manager, and applications that update data in a push model such as our grid coloring demo application discussed in section 4.2.

1.1 The Shared Container Model

Active Containers is based on the shared container model. It implements a specific type of object sharing through the use of a standard container interface. The toolkit allows the programmer to store data structures that must be shared in a *writer* container object. A r*eader* container object can then be created by another processes and registered with the *writer* container in order to share the data. Changes to a shared *writer* container are propagated to all registered *reader* containers, thus providing a distributed shared memory system.

1.2 Active Messages

The Active Containers toolkit takes advantage of a common user-level networking abstraction known as Active Messages [7] in order to provide asynchronous updates to *reader* containers, while achieving reasonable performance. The basic idea behind Active Messages is that the head of a message contains a reference to a user-level handler that will extract the message and integrate it into the ongoing computation. Active Containers make direct use of Active Messages for active container management and shared memory updates.

In order to achieve high portability, the Active Messages 2 API [10] was used for all communication by the toolkit. AM2 is the current standard Active Messages API that has been developed at U.C. Berkeley. For the Active Containers toolkit, a version of AM2 called AMVI was implemented over the Virtual Interface (VI) architecture [8, 9], a high-speed user-level network. The VI architecture is an industry standard that resulted from the Cornell University U-Net project [11].

1.3 Active Containers

Active Containers themselves are objects that implement a standard shared container interface. Although the toolkit requires that all sharable objects implement the standard interface, once this is achieved, using the toolkit is relatively transparent. After an active container is created, subsequent method calls to the shared object are made to the active container.

When a writer's active container function is called in place of the original object, the container passes the function to the original object and sends a message to all registered readers. The toolkit uses function shipping to update readers without sending the entire object after every call.

When an active reader container receives an update, it places the message in a queue for lazy evaluation. In this way, the handler for incoming messages executes fast, simple code and the updates themselves are only performed when needed.

2 Related Work

2.1 Active Messages Implementations

Active Messages were introduced in 1992 [7] as a low-level communication abstraction that was intrinsic to message passing and shared memory machines. Aside from the original implementations on the nCUBE/2 and CM-5 in [7], the active message communication mechanism has been implemented on many different platforms, including the IBM SP-2 [15], Meiko CS-2 [16], Thinking Machines, and the Cray T3D.

The AM implementation over U-Net [11] is closest to ours. U-Net was designed to completely remove the kernel from the critical communication path, without sacrificing protection for individual processes. U-Net contains an Active Message layer that maintains the necessary handlers, performs retransmissions as needed for reliable delivery, and implements a fixed-sized window-based flow control protocol. Messages are received through explicit polling. Our implementation does not need to perform explicit polling to receive messages. Furthermore, by using the current AM2 interface as opposed to the Generic Active Messages (GAM) 1.1 Specification [17], our toolkit can take advantage of the most current standard.

2.1 Distributed Shared Memory Systems

Several techniques have been devised to implement shared memory. One common approach is to use the virtual memory paging system to invalidate or update all copies of a shared page when a write occurs. This approach closely resembles the traditional directory-based hardware solution to shared memory. The groundwork for page-based systems was laid by Li and Hudak [1]. They implemented several protocols to maintain memory coherency using both distributed and centralized memory managers. Their results suggested that a shared virtual memory system could be used on a large-scale cluster where the processes typically share pages for a short period of time. They concluded, however, that the shared virtual memory architecture was unlikely to perform well for data that was updated frequently. *Treadmarks* [2] is one of the more successful systems that followed from Li and Hudak's work. Through a lazy implementation of release consistency, the Treadmarks system reduces the amount of communication necessary to maintain coherency. The lazy aspect of Treadmarks also mitigates the effect of false sharing which can often be a problem due to the coarse sharing granularity of a page-based system. Treadmarks is often used as a performance comparison for other shared memory systems.

Software-based systems, on the other hand, share on the level of software-based memory structures. This approach virtually eliminates false sharing and allows the programmer greater control over sharing granularity. Both the Linda [3] and Emerald [4] systems are examples of this approach.

Orca [5] is a more recent design that has significant advantages over many of its predecessors. Like Emerald, it requires the programmer to use a specially designed object oriented language. Shared data structures are encapsulated in shared data objects and updates are propagated through the use of function shipping. Orca integrates synchronization with object access and automatically determines object placement and replacement strategy. To guarantee object consistency, Orca implements totally ordered group

communication by using a centralized sequencer and history buffers. Our work on Active Containers specifically tried to avoid the use of a non-standard programming language, the need for special compiler support, and the overhead of maintaining object consistency in the presence of multiple independent writers.

The C Region Library (CRL) project [6] is a programming library for sharing regions of memory. Like Active Containers, it doesn't require any special hardware, compiler, or operating system support. Using CRL, the programmer allocates regions of memory that can be shared. In order to use a shared region of memory, the programmer must delimit access to it using CRL annotations. Our work on Active Containers attempts to provide the same toolkit-like approach as CRL, however, by sharing objects rather than regions, our solution is more transparent to the user.

3 Active Messages over VI

The AMVI Active Messages implementation conforms to the Active Messages 2 (AM2) API [10]. AM2 generalizes previous Active Messages interfaces to a broader application base. It provides support for multiple processes using Active Messages for different purposes through the same network resources. Previous implementations of Active Messages had naming models that were sufficient for single-program multiple data parallel programs and an error model based on fail-stop semantics. AM2 attempts to provide a standard naming system sufficient for client-server computing and a more advanced error-handling model for more general applications. The AM2 interface also adds full support for multi-threaded applications and a more standard asynchronous communication event-handling model.

Implementing AM2 efficiently requires network hardware that allows user-level access. The Virtual Interface (VI) architecture [9] was used because it has industry support as a standard for clusters.

The AM2 interface provides a set of over 45 functions for manipulating the Active Messages layer. The majority of these deal with endpoint and bundle management and the sending of active message requests and replies. The entire implementation of AM2 over VI was approximately 6,000 lines of C++ code.

3.1 AM2 Implementation

The AMVI implementation is a user level library based on the AM2 interface. It uses the Virtual Interface architecture based on the interface dictated by the Intel Developer's Guide [8]. The library is written in C++ to take advantage of object oriented design, although our focus was performance and simplicity.

3.1.1 Endpoints

The AM2 interface defines an *endpoint* as the object through which communications occur. Processes can create new endpoints at will that are guaranteed to have unique endpoint names. Any two endpoints within a cluster can communicate with one another.

Endpoints are created as objects that manage both outgoing connections and active message requests through them. Before one endpoint can issue active message request operations to communicate with another endpoint, the programmer must call an API function that *maps* the address of the remote endpoint. The map operation creates a VI connection to the remote endpoint and prepares it for processing. It returns an integer index into the map table for later reference so that the programmer can henceforth refer to communications with the remote endpoint using the index.

Active message request and reply handlers are also registered in a handler table at an endpoint level. Rather then having the physical address of an active message handler at the head of a message, an index into the handler table is used. The handler with index 0 is reserved for returning messages to the sender in the case of an asynchronous error.

3.1.2 Naming

The AM2 specification calls for unique, opaque names for each endpoint created. Any endpoint in a cluster must be able to map any other endpoint in the cluster given its opaque name. In order to support this, AMVI uses the following format for an endpoint name:

computer_name : process_id : bundle : endpoint

In order to connect to a remote endpoint, the endpoint name must translate to a VI address. The computer name is translated to the VI nic address. The process id and the bundle number are stored as hexadecimal fixed-width values and placed into the discriminator portion of the VI address. The VI discriminator is a minimum required length of 16 bytes. It is used for matching in the establishment of client/server VI connections.

The AM2 interface does not specify any sort of name service for Active Messages. In order for applications to easily obtain the opaque endpoint names of remote endpoints, the following extensions were added to the AMVI implementation:

AMNS_Init(…);
AMNS_Lookup(…);
AMNS_Register(…);
AMNS_Unregister(…);

These functions allow a process to register an endpoint's name with a name server so that other processes can perform a lookup based on a well-known registration string. The name service currently runs over VI and makes use of a name server application for storing mappings between registration strings and endpoint names. This could easily be adapted to any other transport medium.

3.1.3 Bundles

The AM2 interface attempts to avoid the common problem of operational deadlock by introducing *bundles*. Please refer to Section 3 in [10] for a complete discussion of the deadlock issues. A bundle is a set of communications endpoints that are created by one process. All endpoints within a bundle are treated as a single unit for communication. Incoming messages for endpoints within a bundle are handled atomically, while incoming messages for endpoints in different bundles are not. By correctly dividing endpoints into bundles that service them, and allowing multiple bundles, it is possible to avoid deadlock situations and keep the number of system resources utilized to a minimum.

AMVI bundles are implemented as objects that contain currently active endpoint objects and work with two threads. The first thread is used for handling incoming connections to endpoints within the bundle and for cleaning up previously closed connections. It continuously sets up a new connection object and waits for an incoming connection. The thread also periodically wakes up and checks a queue of dead connections for any that can be completely retired. A connection can be retired if all of its outstanding active messages have been processed or returned to sender.

The second thread is used to atomically process incoming requests for endpoints within the bundle. It does this by polling a VI completion queue for a completed descriptor. After polling for approximately the VI round-trip latency it then waits on the completion queue in order to relinquish the processor (an examination of the benefits of such a scheme can be found in [12]). Incoming active messages are handed off to the appropriate connection for processing.

3.1.4 Connections and Flow Control

Connection objects are used for communications between endpoints. One of the limitations of the Virtual Interface architecture that is not present in its predecessor, U-Net, is that it is completely connection oriented. For every node in an n node cluster to communicate with every other node, n(n-1)/2 connections are required. It is for this reason that a connectionless architecture such as U-Net would lead to simpler endpoint to endpoint communications.

AMVI uses VI's unreliable delivery communications model. This is recommended over VI's reliable delivery model for extremely low-latency communications. It does not guarantee in-order delivery but does guarantee asynchronous detection of message corruption, empty receive queue, and other catastrophic errors. In order to ensure in-order, exactly once delivery of active messages a window-based flow control scheme is used. This also provides automatic buffering of all outstanding requests and replies, allowing a return to sender in case of error. The fact that outstanding requests and replies must be buffered mitigates the cost of using a window-based scheme as opposed to using VI's reliable delivery model. The window

size on an endpoint is set by the application using the API function AM_SetExpectedResources. This function sets the maximum number of outstanding requests, r, per remote endpoint. Every connection allocates enough buffers to handle 2r active messages (r pending requests and r replies). The AM2 specification requires all requests to be explicitly replied to exactly once, therefore requests are not acknowledged. It also specifies that replies must be returned to the sender in an error condition, therefore AMVI must explicitly acknowledge active message replies when necessary.

3.1.5 Error Model

The AM2 specification provides a significantly more application-friendly error model then previous fail-stop semantics. When an error can be detected synchronously, an explicit error value is returned from the function call. All asynchronous errors are delivered to a special active message handler known as the "undeliverable message handler". Both catastrophic VI errors and errors that occur on the receiving endpoint result in messages being returned to sender. This is taken care of in the AMVI implementation by the automatic buffering of the window-based flow control scheme. Any sent messages that have been explicitly acknowledged are considered retired, while messages still in the send window can be returned to the sender if an asynchronous error occurs.

3.1.6 Sending and Receiving

To send a request message, AMVI copies the message into a pre-allocated send buffer that is already pinned and registered with the VI layer. This message is then placed in the send window and posted to the corresponding connection's VI send queue. If the send window is full then it is up to the application to wait for replies and retry the send. The buffers for messages that are pulled out of the send window are recycled for later use. Sending active message replies work in the same fashion as a request, except that a reply is guaranteed space in the send window.

In order to accommodate incoming messages, receive buffers that have been pinned and registered with the VI layer are posted on a connections VI receive queue. A pointer to the associated connection is hidden in the VI descriptor for the receive buffer. As explained in section 3.1.3, a thread associated with the bundle polls and waits on the bundle's completion queue. When a receive descriptor has completed, it is pulled off the VI's receive queue and the hidden connection pointer is used to route the incoming active message directly to the receiving endpoint. The active message contains an index into the receiving endpoint's handler table. This index is used to find the correct handler routine to execute. When the handler returns, the receive buffer can be reused. In this manner, incoming active messages are handled atomically within a bundle.

It is important to note that a send operation requires one copy of the message, while a receive operation is essentially zero copy. A send-receive operation only requires a single copy.

3.2 AM2 Performance

The main performance metric for AMVI testing was round trip message latency. The tests were performed on a cluster of 450 MHz Intel Pentium II based systems with a 100 MHz system bus running Windows 2000 beta3. The VI nics used in the cluster are Giganet GNN1000 cards connected through a Giganet GNX5000 switch [13].

Figure 1 shows round trip AMVI message times for various data transfer sizes. The round trip latency times represent the total time for a single AM_Request4I operation with the given size payload followed by a single AM_Reply4I operation with the same size payload. This includes the time required to invoke the request handler routine and the corresponding receive handler. The raw VI round-trip message latency for a zero byte message is 17 ?s. In comparison, the AMVI round-trip message latency with a transfer payload of 4 words is 31 ?s. The main cause for the roughly linear increase in round-trip latency with an increase in buffer size is due to the fact that sending an active message requires a single copy operation. This single copy on send cannot be avoided if the AM2 API is followed exactly. Larger message transfers require the use of AM2's "xfer" operations. These perform a copy on send as well as a copy on receive. For a 1024 byte transfer, the round trip latency is 75 ?s.

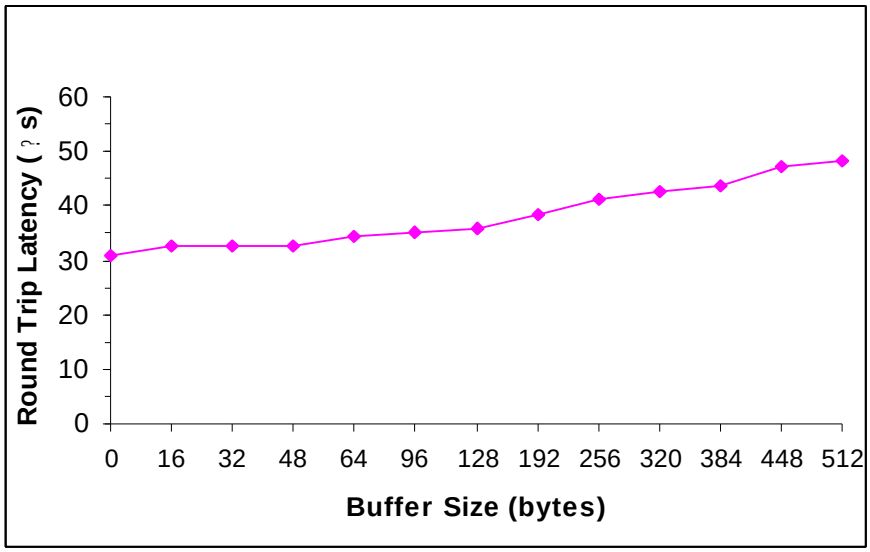

Figure 1: AMVI message round-trip times as a function of message size.

4 Active Containers

Active Containers provides the interface to send and receive updates for a shared data structure. The writer simply creates a writer active container and then routes all method calls through the container as opposed to the original structure. Readers create an active reader container with an empty container object. The reader container manages a queue of received function calls, and executes the calls when a read is issued. An overview of the toolkit is given below in Figure 2. SO refers to copies of the shared object. The Q is a FIFO queue of updates. H is the active message handler.

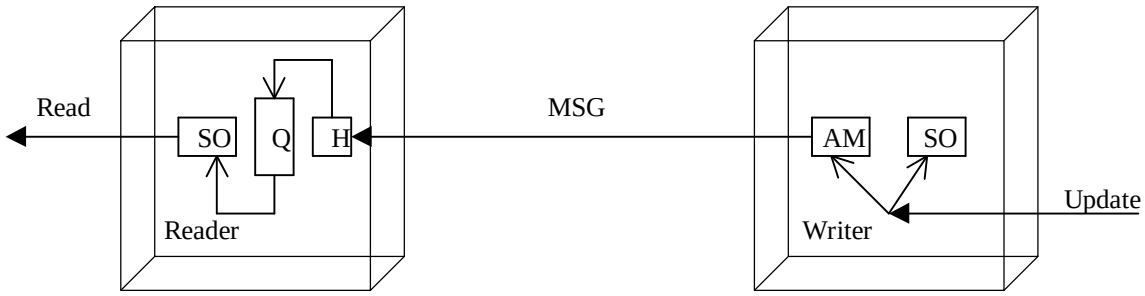

Figure 2: Overview of Toolkit Internals.

The toolkit imposes restrictions on what objects can be shared and which functions can be shipped through C++ inheritance. A sharable class must implement the Object interface and therefore provide marshalling support. Marshalling and demarshaling are done through a stream-based interface.

While the toolkit only provides function shipping for insert, delete, and clear with no more than 2 arguments, adding more functions to the toolkit is trivial.

The toolkit also provides some reliably guarantees. A reader will be notified of a writer's crash within a certain timeout period and vice-versa.

A main goal in the development of Active Containers was to make them easy to use. Through the implementation of several sharable objects, we have verified that this design goal has been met.

4.1 Implementation

The toolkit is comprised of a set of base object types and a factory to create reader and writer active containers. A programmer creates a container object that derives from the standard container class. This container object is then passed to the factory in the creation of either a reader or writer active container. In order to make the contents of the container shared, all access to the container must occur through the newly created active container object. As far as the programmer is concerned, the actual memory sharing is transparent in the use of active containers.

4.1.1 Base Object Types

The *Object* class provides a simple marshalling interface and point of entry for the garbage collector. All objects that make use of active containers must implement the Object interface.

The *Standard Container* class provides a simple container interface. Both reader and writer active containers implement this interface. In order to use active containers, the programmer must derive a container object from the standard container class. An instance of this derived class is passed to the factory for active container creation. The standard container interface itself is derived from the Object class, so standard containers can be marshaled in their entirety and sent over a network.

4.1.2 Data Marshalling

Since the toolkit does not know anything about the structure of the shared object, the toolkit's user must provide marshaling and demarshaling code to allow the objects to be transformed into a buffer of bytes that can be sent in an active message. We created two classes, MarshalStream and DemarshalStream, to lessen this burden for the toolkit's user.

The *MarshalStream* class simplifies the procedure of marshalling objects. An object is marshaled by writing bytes to an instance of the MarshalStream class. The MarshalStream fills an internal buffer with the given input. When the buffer is full, it sends an active message request with the buffer to the recipient. When an object is finished marshaling, the last buffer is sent even though it may not be full. Marshaling completion is detected by the receipt of a buffer that is not completely full. In the rare case that a marshaled object requires an integral number of buffers, the MarshalStream object sends an empty buffer to signify the end of an object.

On the receiver's side, a DeMarshalStream object handles a series of buffers linked together. An object is demarshaled by reading bytes from the DeMarshalStream.

Using these streams, a variable length string can be marshaled with the following code:

```
OBJ_RETURN aString::Marshall(MarshallStream* stream)
{
        int strLength = sizeof(sData);          // first send the size of the string
        int bytesWritten = stream->WriteBytes(&strLength, sizeof(int));
        bytesWritten += stream->WriteBytes(&sData, strLength);
        if (bytesWritten == (strLength+sizeof(int))){return OBJ_RETURN_SUCCESS;}
        else {return OBJ_RETURN_FAIL;}
}
```

The same string can be demarshaled with the following function:

```
OBJ_RETURN aString::DeMarshall(DeMarshallStream* stream)
{
        // demarshall sData
        int strLength = 0;
        int bytesRead = stream->ReadBytes(&strLength, sizeof(int));
        bytesRead += stream->ReadBytes(&sData, strLength);
        if (bytesRead == (strLength + sizeof(int))){return OBJ_RETURN_SUCCESS;}
        else {return OBJ_RETURN_FAIL;}
}
```

4.1.3 Garbage Collection

Garbage collection becomes a requirement due to the use of function shipping. When a writer inserts an object, the toolkit must ship copies of that object to each reader. Similarly, when the writer deletes an object, the corresponding object must be deleted from each reader. The difficulty arises when the writer removes an object from its shared structure, but does not delete the object. This can happen for various reasons. For example, a writer may have two structures referring to the same object, but only one of the structures is shared. In this case, a writer does not want to delete an object, but the readers do.

Garbage collection in the toolkit is currently handled by the Boehm-Demers-Weiser conservative garbage collector [14], which is freely available. It uses a mark and sweep algorithm and was easily integrated into our toolkit. All objects that are derived from the *Object* class are automatically allocated and tracked by the garbage collector.

4.1.4 The Active Container Factory

The factory creates reader and writer containers and prepares them for use. It also manages the AMVI layer. In this manner the programmer does not have to be concerned with the underlying active container creation and the process by which reader active containers register and unregister themselves with writer active containers. Instead, the programmer only needs to know the string that was used to name a writer active container when the writer was created using the factory.

When a writer is created, the factory registers the writer's AMVI endpoint with the name service and encapsulates the given object to be shared. When a reader is created, the factory queries the name service for the requested writer's endpoint and registers the newly created reader active container with the writer so that it will send updates to the new reader. Finally, the factory jump-starts the reader by requesting the writer to send a marshaled representation of the shared container object.

The factory also deals with active container deletion. Before a writer active container is deleted, all reader active containers are notified and the writer's AMVI endpoint is unregistered from the name service. Before a reader active container is destroyed the factory notifies its corresponding writer active container.

4.1.5 Writer Active Containers

The writer active container maintains an array of endpoints corresponding to registered readers and a pointer to the shared container object. The programmer is expected to delegate all accesses to the internal shared container through the writer active container. For each function that the toolkit supports, the writer active container calls that function on the internal shared object and ships the function to all registered reader active containers. It does this by marshaling the function arguments and sending an appropriate active message update request to each of the readers. The writer also marshals and sends the complete shared container object when requested by a reader.

In order to provide some reliability guarantees the active writer container also maintains a thread that sends a ping request to each reader after a certain timeout period in which no update has been sent.

4.1.6 Reader Active Containers

The reader active container maintains a pointer to an object that has been derived from the Standard Container class. The user must provide this object when the reader is created. This requirement is a natural consequence of both marshaling and function shipping. Function shipping can only work if the reader and writer agree on what actions a function performs. Similarly, the readers and writer must agree on what data members are being manipulated to demarshal the objects correctly. One limitation of function shipping is that updates that do not alter the shared object are still sent to the readers. This cannot be avoided in a transparent manner without significantly reducing performance.

In addition to the user provided object, the reader maintains a queue of function calls and their corresponding arguments. When a shipped function is received, an AM handler puts the message on the queue and a reply is sent to the writer to verify receipt of the message. When the user performs a read operation, the arguments for the functions in the queue are demarshaled and the functions are executed in FIFO order. The underlying message system insures that no messages are dropped and that all updates are received in the order in which they were sent. Therefore, when all of the function calls are executed the reader's con-

tainer has the same information that the writer has up to the point of the last received update. Since a ping or an update is sent periodically, the read active container can never fall far behind the writer active container.

Maintaining the queue increases the toolkit's performance. First, because the handler only has to add buffers to the queue rather than demarshal and execute functions, the handler can be simple, fast code. Since AM handlers are non-blocking, this is essential. Secondly, the lazy evaluation of shipped functions leads to a penalty that is only incurred at read time. If reads are done more frequently than writes, then the queue is kept fairly empty and a read requires very little time to update the reader's version of the shared object. If reads are performed infrequently, then the queue may fill up between reads. When the queue is too full for another update, the writer is asked to send the entire object and the queue is flushed. In this manner an infrequent reader may skip several intermediate function evaluations. By changing the queue's size, the toolkit can be optimized for best performance in a particular situation. The combination of function shipping and sending complete objects minimizes the size of messages without requiring complex handlers, infinite queues, or inefficient updating.

4.1.7 Reliability Guarantees

The toolkit uses pings and replies to insure that readers and writers are alive. If an update has not occurred within a specified period of time, then the writer pings each registered reader to show that it is still alive. When a reader receives either an update or a ping, it sends a reply to the writer. If the writer does not receive a reader's reply after a timeout, the writer assumes that the reader has died. Similarly, if the reader does not receive an update or a ping after a timeout, the reader assumes that the writer has died.

If the reader is incorrectly thought to be dead, then that reader will not receive the writer's next update and will consequently assume after a certain timeout period that the writer has died. Similarly if the writer is incorrectly thought to be dead, then the reader must not have received the writer's last message. Therefore, the reader will not have responded to the writer and the writer will believe that the reader has died. In either case, when one component is believed to be dead, they both agree that the connection is broken.

If the reader's queue is compromised or corrupted in a noticeable way, then the reader can recover by simply asking the writer to send a copy of the entire shared object.

4.2 Usability

To test the usability of the toolkit, we built a number of shared container objects. The only work necessary to create a shared container is to implement the shared container interface. All data objects stored in a shared container must implement the Object interface and therefore have to provide marshaling/demarshaling code as well.

For example, we were able to modify a standard demo application to make use of active containers in an afternoon. The demo was originally created to demonstrate the total ordering properties of a transport protocol. The demo keeps track of a grid of squares, each containing a number and having a certain color. When several copies of the demo are run concurrently and on different nodes, the colors and values of the squares in the grid should remain the same on all nodes.

In order to turn the demo into an active containers demonstration, we simply had to make the grid an object that implements the standard container interface. We then had to create wrappers derived from Object for the data values that were to be inserted in the container. One node acts as the writer and creates a writer active container based on the grid object. This node continuously updates its grid object through the active writer container. The other nodes create reader active containers, supplying them with a grid object. All nodes continuously read the grid values from their active containers and display the results on the screen. The demo application provides visual confirmation that writer updates are received by several readers quickly. The time for an update to be sent by the writer, received by the reader, and placed in the queue is approximately 18?s.

Active containers are also fairly easy to extend. If the shared container object uses functions that the toolkit does not support, then the writer must send an entire updated object whenever the unsupported function is called. If the active reader container does not know what function was shipped, then the container cannot call that function on the reader's version of the shared object. Fortunately, it is very easy to add extra functions to the active containers. The additional functions must be mapped to a unique integer.

When a message with that integer is processed the corresponding function must be called in the reader's version of the shared object.

5 Future Work

5.1 Active Messages over VI

The most promising future work with active messages over the Virtual Interface architecture would be pre-registration of message buffers. This would require changes to the AM2 interface. One area in which this would be an improvement would be to add an extension to the AM2 interface to allow applications to pre-specify send buffers. This would allow zero-copy active message request and reply operations since the buffers would be known to the active message layer and registered with the VI layer in advance. Currently, outgoing active messages require a single copy into a pre-registered buffer that resides in the AMVI layer.

Another possible area of future work would be to eliminate copying, during AM2 "xfer" operations. These currently allow for any data buffer to be transferred to an offset in a remote node's virtual memory data segment. The VI architecture supports remote DMA write operations on a pre-registered remote memory address using a specific memory handle that is provided at registration. Using RDMA write operations for "xfers" would enable zero copy on the receiving side. This combined with allowing the application to pre-register send buffers would enable "xfer" operations to be zero-copy in their entirety. The AM2 interface, however, currently supports writing to any offset in the virtual memory segment. In order to use RDMA write operations, the sender first has to request that the remote endpoint register its virtual memory segment at the given offset and reply with a VI memory handle. Only then can the RDMA write operation occur. A solution to this problem would be to only allow xfer operations to occur at predetermined offsets within the remote endpoint's virtual memory segment. These offsets could be pre-registered and the corresponding handles cached on the sending endpoint. The above is not a problem for "Get" operations because a memory handle can be sent in the initial "Get" request and on the fly registration can be performed if necessary.

A third possible area for future work is in modifying the AM2 API to have a more specific error model. Currently the API supports six possible AM function return values. In implementing the AM2 interface it quickly became apparent that the error return codes often do not apply to the given error condition. To make matters worse, the API does not explicitly state what error return codes may be returned by a function and for what reason. A more rigorous API specification would eliminate this problem.

5.2 Active Containers

Perhaps the best area for future work with active containers would be to port it to Java. Java's Serializable interface would remove the need for user's to write their own marshaling code. Garbage collection would also cease to be a concern. It is easy to imagine taking all of Java's "java.util" data structures and turning them into active containers in addition to providing a more generic active container interface.

In continuing with the C++ implementation of active containers, additional functions could easily be added to the containers. The functions provided in the Standard Template Library would be a good set to add to active containers.

Another possible improvement would be to add a low priority thread to the reader. This thread would work on processing commands in an active reader container's queue in the background to reduce the frequency of complete object updates.

A final area of future research for active containers would be in relaxing the single writer constraint. This could be done by forcing some sort of total ordering on updates or by maintaining a single writer, but allowing that writer to change.

6 Summary

This paper introduced active containers, a toolkit for implementing multiple reader, single writer shared memory objects. Shared memory updates are propagated using an implementation of AM2 over the Virtual Interface architecture. Our measurements of round-trip communication times substantiate the high performance of the AMVI layer. The usability of the toolkit was examined in the development of various shared objects. While we are pleased with the Active Containers toolkit, there is still great potential for continued research in this area.

Acknowledgements

We thank Werner Vogels for adopting us into the Cornell cluster research group and allowing us to use the giganet cluster for testing. We would also like to thank Thorsten von Eicken for teaching the cs614 class. Finally we would like to thank Robert Tsai, Dan Dumitriu, and our other friends in the systems lab for their support.

References

1. K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems." *ACM Trans. Computer Systems* Vol. 7, No. 4. Nov. 1989. pp. 321-359.

2. A. Cox, S. Dwarkadas, P. Keleher, and W. Zwaenepoel. "TreadMarks: Distributed shared memory on standard workstations and operating systems." *Proceedings of the Winter 94 Usenix Conference*. USENIX Assoc., Berkeley, Calif. pp. 115-131

3. A. Deshpande and M. Schultz, "Efficient Parallel Programming with Linda," *Proceedings of the 1992 Conference on SuperComputing*, 1992, pp. 238-244.

4. E. Jul, H. Levy, and A. Black, "Fine-Grained Mobility in the Emerald System, " ACM Trans. Computer Syst., Vol. 6, No. 1, Feb. 1988, pp. 109-133.

5. Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Ruhl, and M. Frans Kaashoek, "Performance Evaluation of the Orca Shared Object System", *ACM Trans. Computer Systs*., Vol. 16, No. 1, Feb. 1998

6. Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. "CRL: High-Performance All-Software Distributed Shared Memory." *SIGOPS*. Dec. 1995 pp 213-228.

7. Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *The 19th Annual International Symposium on Computer Architectur*e, May 1992, pp. 256-266.

8. "Intel Virtual Interface (VI) Architecture Developer's Guide, Revision 1.0, Sept. 9, 1998." ftp://download.intel.com/design/servers/vi/intel.pdf

9. "Virtual Interface Architecture Specification, Version 1.0." Dec. 1997. http://www.viarch.org/

10. Alan Mainwaring, David E. Culler, "Active Messages Applications Programming Interface and Communication Subsystem Organization", Draft Technical Report, Computer Science Division, University of California at Berkeley

11. Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", *Proc of the 15th ACM Symposium on Operating Systems Principles*, December 3-6, 1995.

12. Andrea C. Arpaci-Dusseau, David E. Culler, Alan Mainwaring, "Scheduling With Implicit Information in Distributed Systems", *Sigmetrics'98 Conference on the Measurement and Modeling of Computer Systems*

13. Giganet, Inc. http://www.giganet.com/

14. Hans Boehm, Alan Demers, M. Weiser, *Boehm-Demers-Weiser Garbage Collector*, http://reality.sgi.com/boehm_mti/gc.html

15. C-C. Chang, G. Czajkowski, and T. von Eicken, "Design and Implementation of Active Messages on the SP-2", *Cornell CS Technical Report*, February 1996, pp *96-1572*.

16. Klaus Schauser and Chris Scheimann, "Experience with Active Messages on the Meiko CS-2", *Proc. of the 9th International Parallel Processing Symposium*, April 1995.

17. David Culler, et. al., "Generic Active Message Interface Specification, version 1.1", http://now.cs.berkeley.edu/Papers/Papers/gam_spec.ps