Using Active Messages to Support Shared Objects

Leendert van Doorn
Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

This paper discusses a reliable group communication system using active messages to update shared objects. We discuss the model, implementation techniques, and our preliminary performance results.

1. Introduction

The performance of parallel programming systems on loosely-coupled machines is mainly limited by the efficiency of its message passing communication architecture. Rendezvous and mail-boxes are the traditional communication mechanisms upon which these systems are built. Unfortunately both mechanisms incur a high latency at the receiver side between arrival and final delivery of the message.

An alternative mechanism, active messages [6], reduces this latency by integrating the message data directly into the user-level computation as soon as it arrives. The integration is done by a user specified handler, which is invoked as soon as possible after the hardware receives the message.

For interrupt-driven architectures the most obvious design choice is to run the handler directly in the interrupt service routine. This raises, however, a number of problems: protection, possibility of race conditions, and the possibility of starvation and deadlock. Consequently the handler cannot contain arbitrary code or run indefinitely.

In this position paper we describe the initial design, implementation and performance results of a group communication system on top of Amoeba [4] using active messages to efficiently update shared data-objects. We also propose a technique for generalizing the active message concept.

2. Object-based Group Active Messages

The shared data-object model [1] provides a powerful abstraction for simulating distributed shared memory. Instead of sharing memory locations, objects with user defined interfaces are shared. Objects are updated by invoking operations via their interfaces; the details of how these updates are propagated are hidden by the implementation. All operations on an object are serialized.

Shared objects are implemented using active replication. To do this efficiently, we have implemented a group communication system using active messages. In our implementation, a the run-time system associates a mutex and a number of regular and special operations with each object. These special operations are invoked by sending an active message. They are special in that they must not block, cause a protection violation, or take longer than a certain time interval. They are executed in the network interrupt handler and run to completion once started. This means that they are never preempted by other active messages or user processes. When the mutex associated with an object is locked, all incoming active messages for it are queued and executed when the mutex is released. Therefore active message operations do not need to acquire or release the object's mutex themselves since it is guaranteed to be unlocked at the moment the operation is started.

Active message invocations are multicast to each replica of the shared-object. These multicasts are totally-ordered and atomic. That is, in the absence of processor failures and network partitions it is guaranteed that when one member receives the invocation, all the others will too, in exactly the same order.

Associating a lock with each object is necessary to prevent an active message from starting an operation while a user operation was in progress. Active message operations are bounded in execution time to prevent deadlock and starvation.

There restrictions placed on active message handler are currently expected to be enforced by the compiler and the run-time system. Section 6 describes a scheme which relaxes these assumptions.

3. Implementation

Each replica of the shared object is registered at the kernel under a unique object name together with an array of pointers to its operations. To perform a group active message operation a member multicasts an invocation containing the object name, the operation to be performed (an index into the object's interface array), and optional arguments.

The multicasting is performed by a sequencer protocol that is akin to Amoeba's PB protocol [3]. In a sequencer protocol one member is assigned the special task of ordering all messages sent to the group. The main difference is that in our new protocol, the individual members maintain the history of messages they sent themselves instead of the sequencer to make the sequencer as fast as possible.

Our implementation takes advantage of the underlying hardware multicasting capabilities. For efficiency reasons, we have limited the size of the arguments to fit in one message.

When a network packet containing an invocation arrives at a machine, it is dispatched to the active message protocol code using the standard Amoeba facilities. These save machine registers, examine device registers, and queue a software interrupt, which in turn calls our protocol dispatcher. This routine does all of the protocol processing. Once it has established that the invocation is valid it checks the associated mutex. If this mutex is locked, it queues the request on a per-object queue in FIFO order. If the mutex is not locked the dispatch routine maps in the context of the process to which the object handler belongs and makes an up call to it. Whenever an object's mutex is released its lock queue is checked.

4. Experience

We have evaluated the performance of our implementation on a collection of 50MHz SPARCs connected by 10 Mb/s Ethernet. We measured the time it took to multicast a null message from a member to the whole group, and the time it took to handle an invocation. Each member sent exactly 10,000 active message invocations.

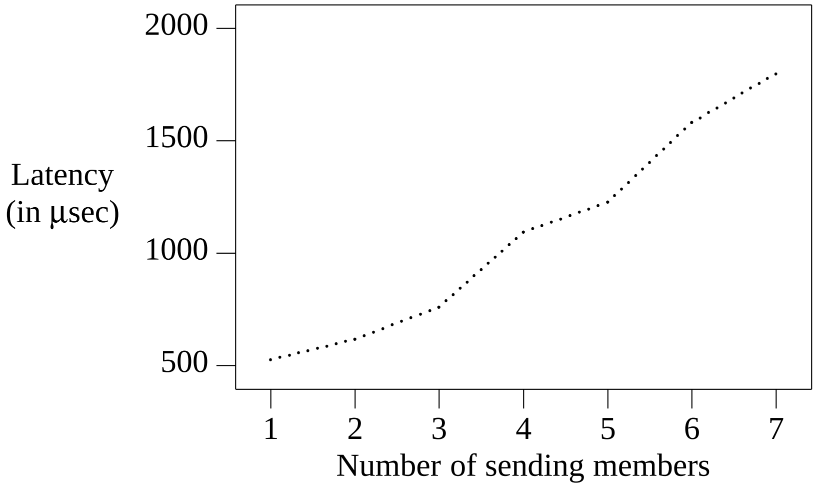

Figure 1. Message latency

Figure 1 shows the latency for sending an invocation to a group. The simple case is a group with two members and one sender, where the sender does not reside on the sequencer machine. In this case the sending member will send a message to the sequencer which will then multicast it. In the current implementation, the member who sent the message waits for it to be sequenced before it sends another one. The 538 µsec figure is thus the cost it takes to get into the kernel, send the message, interrupt the sequencer machine, sequence and multicast it, and then handle the reply on the member machine and return to the application. This waiting for an acknowledgement also causes the initial low throughput as shown in Figure 2 where there is only one sending member.

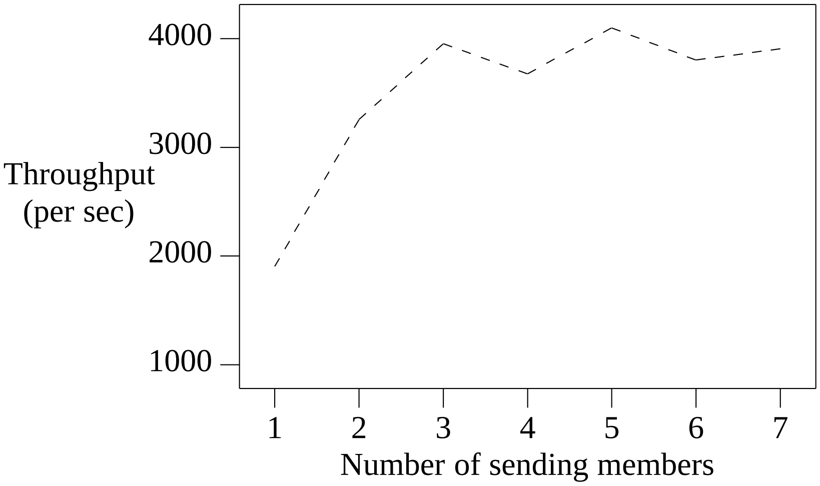

Figure 2. Sequencer throughput

In Figure 2 we have measured the throughput of the sequencer by multicasting null updates and varying the number of sending processes. The sequencer itself, although a member of the group, did not send any messages. As can be seen the throughput improves dramatically as the number of sending members increase and levels off at about 4000 invocations per second. This is about 90% of the maximum throughput for this particular hardware configuration [5]. The increase in throughput can be explained by the fact that members can get to the sequencer and send their message while others are waiting for their acknowledgement.

5. Comparison

Our work combines objects, group communication, and active messages to efficiently support shared data-objects. Our group communication protocol is akin to the Amoeba group communication protocols [3], but does not use the sequencer for recovery. In the Amoeba group protocols the sequencer buffers all the messages sent to the group and members turn to the sequencer to recover. This requires the sequencer to do history management and copying of messages out of the network hardware driver buffers, which slows it down.

Von Eicken's active messages [6] are like our invocations. In von Eicken's work a message contains an address of arbitrary code to be executed. Our scheme is more restrictive in that each member has to specify what operations the client may invoke on an object. We believe our method is better structured and offers better security as only official ‘‘special operations’’ may be invoked by an incoming message.

Von Eicken's active messages are also executed in the network driver's interrupt handler and should run to completion and not block. Unlike our scheme, however, they do not provide a mechanism for dealing with concurrency control. A different approach is taken by Hsieh et al. [2] where active messages are generalized and continuations are used when they are about to block.

6. Open Questions

The main problem with active messages is that of the incoming message handler. Can an active message operation block, how to prevent it from causing protection violations, and how long can it execute? In our current implementation active message handler are user specified interrupt routines which cannot block or cause a protection violation and should run to completion with a certain time interval. In our model these restrictions are expected to be enforced by the compiler and the run-time system.

A more general view is conceivable where user-level handlers have no limitations on the code or on the time to execute. One possible implementation is the use of continuations [2] whenever a handler is about to block. However, with continuations it is hard to capture state information and dealing with exceeding execution quanta is tedious.

Another possible implementation is to create a kind of proto-thread for the active message handler. This proto-thread can be turned into a real pop-up thread when it is about to block or when it runs out of time. The proto-thread is created automatically by means of the processor's interrupt mechanism. Every network device has its own page of interrupt stack associated with it. Initially the handler executes on this stack and when it is turned into a real thread it inherits this stack and the network device's interrupt stack is replaced by a new page. This requires a reasonable number of preallocated interrupt stack pages. When we run out of these we drop incoming messages and rely on the active message protocol to recover. Protection in the scheme is handled by mapping in the message handler's object code, instance data, and the stack on which the handler is currently executing.

The advantage of this scheme is that it does not rely on the compiler and run-time system for protection and the handlers themselves have no restrictions on execution time or whether or not they block.

References

1. H. E. Bal, *Programming Distributed Systems*, Prentice Hall, Englewood Cliffs, NJ, 1991.
2. W. C. Hsieh, K. L. Johnson, M. F. Kaashoek, D. A. Wallach and W. E. Weihl, Efficient Implementation of High-Level Languages on User-Level Communication

Architectures, MIT/LCS/Tech. Rep.-616, May 1994.

3. M. F. Kaashoek and A. S. Tanenbaum, Group communication in the Amoeba distributed operating system, *Proc. of the 11th IEEE Symp. on Distributed Computer Systems*, Arlington, Texas, May 1991, 222-230.

4. A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen and G. van Rossum, Experiences with the Amoeba distributed operating system, *Communications of the ACM 33*, 12 (Dec. 1990), 46-63.

5. C. A. Thekkath and H. M. Levy, Limits to Low-Latency Communication in High-Speed Networks, *ACM Transactions on Computer Systems 11*, 2 (May 1993), 179-203.

6. T. von Eicken, D. E. Culler, S. C. Goldstein and K. E. Schauser, Active Messages: a Mechanism for Integrated Communication and Computation, *Proc. of the 19th International Symp. on Computer Architecture*, Gold Coast, Australia, May 1992, 256-266.