Hyperactive Messages

KRISHNA KAVI
The University of Alabama in Huntsville

ALINA HERNANDEZ and DAVID LEVINE
The University of Texas at Arlington

ABSTRACT

Hyperactive Messages are similar to Active Messages; but they are designed with two goals. Like active messages, Hyperactive Messages facilitate efficient communication among computations distributed over a network of workstations. But they also facilitate for the dynamic insertion of code or upgrades to existing code into a computation. If a Hyperactive Message carries data it is processed as a normal Active Message. On the other hand, if a Hyperactive Message carries the code for a service, the code is loaded into the process's address space and linked to an idle thread. With Hyperactive Messages any instance of a distributed program has the capability to dynamically change its configuration by attaching or detaching services. Such a capability can be used for load balancing and dynamic service upgrading.

Keywords: Active Messages, Threaded Abstract Machine, Code-Migration.

1. INTRODUCTION

Hyperactive messages permit for code migration by packaging the executable object with a messge and the code is loaded into the address space an idle thread at receiving node. With Hyperactive messages any instance of a distributed program has the capability to dynamically change its configuration by attaching or detaching services as needed. Such a capability can be used for load balancing and dynamic service upgrading. Our research is related to numerous projects on the design of network of workstations, as well as with research on CORBA, Java and other code-migration projects [4]. However our research goals and model differs from these. Our model is based on three main concepts: fine-grain parallelism, Hyperactive messages, and Associative Broadcast. Fine-grained parallelism is achieved by using the Threaded Abstract Machine (TAM) [3]. In this paper we will describe an implementation and our experience with Hyperactive Messages. The third component of our research is described elsewhere in literature [1].

2. HYPERACTIVE MESSAGES (HAM)

The architecture for implementation of Hyperactive messages is the Threaded Abstract Machines [3]. A TAM program is a collection of code-blocks, which typically represent a function or loop body in a high-level language program. A code-block contains declarations, inlets, and threads. A TAM thread is a non-blocking sequence of instructions. An inlet is a sequence of instructions that handles a specific message and forms the basis for the implementation of Active (and Hyperactive) messages.

Our Hyperactive messaging system requires an implementation of active messages and the ability to dynamically bind computations to threads. We used the GNU gcc compiler with dynamic loading features called lazy loading. By enabling lazy loading, the system's "dynamic library loader" is able to load shared libraries in the program's address space during execution time. To utilize this facility, all dynamically loadable functions must be coded as regular C functions. Once the functions are coded, the source file must be compiled as a dynamic linkable library (DLL) by using the "-shared" option of the compiler. Additionally, the program that requires the loading of the functions must include the "dlfcn.h" header file in its source file, and must be compiled with the "-lmld" option of the compiler. Once the program has been compiled and running, it can use the "mld" interface to load routines in the process's address space, invoke the previously loaded routines or unload routines when they are no longer required. The key "mld" API's can be found in [5].

We designed and implemented Active messages along the Generic Active Messages (GAM) interface definitions provided in [2]. We use UDP/IP for communication among HAM processes (representing a TAM activation frame). The GAM API's can be found in [5]. As in the TAM execution model, HAM messages not only transfer data but also control. Each active message sent must specify the name of a remote inlet responsible for handling the message. Additionally, each message may specify the name of a local inlet to receive acknowledgments. Hyperactive messages are directed to predefined special inlets.

3. HAM LIBRARY PROGRAMMING MODEL

Our initial goal is to prove the feasibility of implementing the Hyperactive messages concept. To prove this concept we implemented a set of library functions that can be used by the parallel versions of conventional C programs. As described in the previous section, we developed a TAM like execution environment and Generic

This research is supported in part by the following grants from NSF, MIP-9796310 and EIA-9729889.

Active Messages using UDP/IP. The current system runs on a network of DEC Unix systems, with GNU dynamic link loaders and Pthread libraries. HAM programs are viewed as sets of inlets, static threads and dynamic threads. More detailed information on the HAM API's can be found in [5].

Inlets. Similar to TAM, out Inlets are nothing more than functions in the user's program that are responsible for handling specific messages.

Static Threads. Static threads, implemented as pthreads, are created at the beginning of the program's execution. When a static thread is created, it waits on a condition variable until an inlet or some other thread forks it. Functions comprising the static thread must be specified and bound permanently. The function body must be wrapped by special instructions called "st_init" and "st_exit". Except for thread signaling, inlets and threads communicate with each other through global variables.

Dynamic Threads. Dynamic threads are similar to static threads, except that the function comprising the thread execution is bound dynamically and the user needs to specify only the number of dynamic threads needed. Dynamic threads are pthreads that wait on a condition variable until a service is attached to them (through a Hyperactive message) and they are forked either by an inlet, another thread, or a remote process. Dynamic threads cannot communicate with other threads and inlets through global variables. The *dynamic_thread_fork* function must be invoked with the following arguments: service name, a buffer containing data to be passed to the thread as argument, and the buffer's size. When enabled, the currently bound thread function (service) is invoked passing the data buffer and size as arguments. The binding of services associated with dynamic threads can be deleted, making the threads available for new bindings.

3.1.Message Types and Message Handling

HAM messages can be of four different types: active messages, Hyperactive messages, dynamic_thread_fork messages, or dynamic_thread_delete messages. All messages are treated as active messages; that is, messages must specify the name of a handler or inlet which will be invoked when the message arrives to its destination. The HAM library contains a generic message handler that's triggered by the SIGIO signal when a packet arrives. Once a packet arrives the SIGIO signal is temporarily disabled, and the handler will read all message packets, then the SIGIO signal is re-enabled. HAM's generic message handler is responsible for: reading the arriving packets, sending an ACK packet back to the sender, buffering packets until the message is reassembled, and invoking a message handler based on the message type.

Active messages are handled simply by invoking an inlet function.

Hyperactive messages. When a Hyperactive message is received, the special HAM handler searches to see if a previous binding for the service name exists, in which case the old binding is replaced by a new binding. If not, an idle dynamic thread is located and a new binding is made. Note that the DLL comprising the new binding must be supplied as a file: "service_name.so". If the message contains the actual binaries, a file will be created by the handler. Subsequently, the code is loaded and the new service is registered. An acknowledgment is returned to the sender if a remote inlet is specified.

Dynamic Thread Fork messages are used to fork dynamic threads. Note that these messages are needed to fork dynamic threads.

Dynamic Thread Delete messages are used for removing an existing binding with a dynamic thread. An acknowledgment is returned to the sender if a remote inlet is specified.

3.2. Library Usage

To be able to make proper use of the HAM library implementation, the user must complete the following steps:

1 Write the program using HAM user's interface [5]. When coding the program the user must remember to:

- Structure the program in such a way that some user functions may act as inlets, and others may act as static threads.
- Use the keyword "i_args" as the only argument to all inlet functions.
- Use the keyword "t_args" as the only argument to all static thread functions
- Use the keywords "st_init" and "st_exit" to enclose the body of each static thread function.
- Declare static thread functions and inlet functions of type void.
- Add p_init(), in_create(), st_create(), and p_exec() to the program's "main" function. The only requirement related to the order of invocation is that p_init() must be invoked first, and p_exec() must be invoked last. If the program does not contain static threads or inlets, in_create() and st_create() may be omitted.

2 Compile the program using the HAM library.

3 Create a configuration file called "executable_name.cfg" in the same directory where the executable was created. This file must have the following format:

- All comments must be preceded by a pound sign "#".
- The first line must have the starting address of a range of available UDP/IP ports, where the number of available ports must be equal to the number of process instances being created. The same set of ports must be available in each of the hosts that will be used to run the parallel program.
- The second line must contain the number of hosts that will be used, and the number of process instances that will be created. These two numbers must be separated by a comma.
- The following lines are reserved for the names of the hosts that will be used to run the parallel program. Each

line may contain only one host name, and the names used must match those located in the "/etc/hosts" file.

A typical a configuration file should look like.

```
# Base port address
7000
# Number of Hosts, Number of processes
2,7
# Hosts' names
mark
ed
cross1
```

4 Set the "HAM_DLL_DIR" environment variable so it points to the directory where all the dynamic loadable libraries (DLLs) are located. All process instances will try to locate the DLL libraries using this variable. If this variable is not defined the processes will assume they can open the file without specifying a path. Therefore, it's advisable to add the instruction to set the "HAM_DLL_DIR" environment variable to the user's ".cshrc" file in all the hosts involved in the execution of the parallel program. The DLL libraries contain the services that may be sent as Hyperactive messages. The user must remember that a library that contains a service called, for example, "hello" must be stored in a file called "hello.so" for a process to be able to read it.

5 Make sure that the user can execute remote commands (i.e., "rsh" and "rcp") from any host to any other host, without specifying a different login name.

6 Use the "ham_run" program to create all process instances in the different hosts. "Ham_run" is a very simple program that uses "rcp" and "rsh" commands to create the process instances based on the information present in the program's configuration file. The output from the processes are redirected to files named "executable_nameprocess _id.out". For example, if the name of the executable is "hello", and the process's ID is zero the output of the process is redirected to "hello0.out". All processes are created from the user's home directory and the output files are created in the same directory.

3.3. An Example Scenario

In order to show how HAM threads work, Figure 1 shows a simple example that involves the dynamic binding, deletion and rebinding of a service to a remote dynamic thread.

First, p_0 sends to every process p_i, where i = 0,..P-1 (in our case i=2), two hyper-active messages: one containing a service that prints "Hello WORLD" and sends an active message back to the process that forked the service; and the other containing a service that prints the local time using the format HH:MM:SS.

After receiving acknowledgment to the Hyperactive messages, p_0 will send a new Hyperactive message to all the processes with a new version of the "Hello WORLD" program, that will print "Hello UTA". This involves the deletion of previous binding and association of a new binding. Once p_0 receives acknowledgment, it will repeat the same steps to upgrade the service which this time will print "Hello UAH".

In the second half of the experiment, p_0 sends a Hyperactive message containing the code for a service that will read the operating system process ID and send it back to the process that forks the service. Subsequently, it sends a Hyperactive message that contains the code for a service that simply prints "Testing dynamic thread table load". Dynamic_thread _fork messages are used to execute the service and receive the OS process ID's. p_0 proceeds to print them.

Since, the processes are only capable of holding three dynamic threads at one time, all process should reply to p_0 indicating that the second service sent wasn't loaded successfully because of lack of space in the dynamic thread table. Once p_0 receives a reply from all processes indicating the failure in loading the hyper-active message, it will then send a new hyper-active message to all p_i, where i = 0, 1, containing a print time service that replaces the existing one. The new print time service prints the date and time using the following format: WKD MON DAY, YY HH:MM:SS.

At this point in the program's execution, p_0 will sleep for 10 seconds and then sends an active message to all the processes, including itself, requesting that all threads are canceled. Canceling all the threads causes all processes to finish their execution.

4. CONCLUSIONS

In this paper we described a new concept called Hyperactive Messages and its implementation as a feasibility study. We used Generic Active messages and TAM model as a general framework for the implementation. The actual code was developed on a network of DEC Alpha workstations running DEC UNIX, and connected by Ethernet. We use Pthreads for the implementation of HAM threads and inlets, and UDP for communication over the network. We developed and tested several test programs to utilize the capabilities of Hyperactive messages. In this implementation, performance was not our primary concern. However, we hope to revise the implementation to achieve better performance.

Active Messages require message handlers not to block, in order to avoid deadlocks, and to run only for a short period of time, so not to congest the network. In our HAM library, the user must deal with these issues, and must try to avoid these problems by writing short message handlers and avoiding the use of locks within message handlers. Optimistic Active Messages [8] achieve the performance of Active Messages by allowing user code to execute in a message handler instead of a thread, therefore avoiding thread management overhead and data copying time. Handlers are compiled with the assumption that they run without blocking and complete quickly enough to avoid causing network congestion. At run-time this assumption is verified, and if it is false a separate thread is created for the handler. If most handlers neither block nor run for too long

Optimistic Active Messages achieve the performance of Active Messages; on the other hand, if a handler does run for too long the creation of a thread for the handler prevents it from backing up the network. Furthermore, this approach frees the programmer from the burden of dealing with the restriction of Active messages. We may consider adapting similar methods in our future implementations.

5. REFERENCES

[1] B. Bayerdorffer, "Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems", PHD Dissertation, University of Texas at Austin, Dec. 1993.

[2] D. Culler, K. Keeton, C. Krumbein, L. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa, "Generic Active Message Interface Specification", Draft Technical Report, Computer Science Division, University of California at Berkeley.

[3] D. Culler, S. Goldstein, K. Schauser, and T. Von Eicken, "TAM-A Compiler Controlled Threaded Abstract Machine", Journal of Parallel and Distributed Computing, 1993, pp. 347-370..

[4] A. Fuggetta, G.P. Picco, and G. Vigna "Understanding Code Mobility", IEEE Trans. on Software Engr., May 1998.

[5] A. Hernandez. "Hyperactive Messages", MS Thesis, Dept of CSE, UTA, Arlington, TX 76019, Dec. 1998.

[6] R. Stevens, "Unix Network Programming", Prentice Hall, Englewood Cliffs, N.J., 1990.

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

[8] D. Wallace, W. Hsieh, K. Johnson, M. Kaashoek, and W. Weihl, "Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation", PPoPP '95, July 1995.

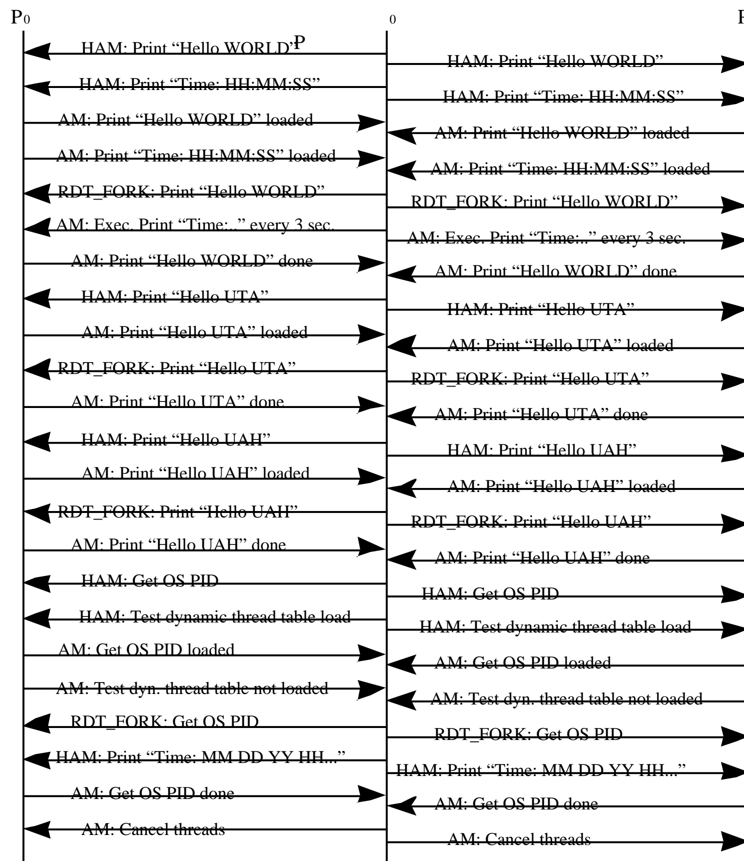

Figure 1. Message flow of the dynamic service upgrade program.