Massachusetts Institute of Technology — Lecture 15
6.892: Theory of Parallel Systems — April 8, 1997
Professor Charles E. Leiserson — Scribe: Mehul A. Shah

Lecture 15: Active Messages

1 Introduction

Active Messages are a primitive communication mechanism which can be used to implement traditional parallel programming paradigms such as shared memory or MPI. It is a protocol layer intended to lie close to the hardware and provide a flexible abstraction to allow programming. The basic concept behind Active Messages is simple. Messages are passed with a header indicating a user-level routine which extracts and integrates the data from the network. At this level, the messages are not buffered. The simplicity and efficiency of this protocol makes it more attractive than most parallel message passing protocols.

First, we examine the typical network interface hardware upon which Active Messages is implemented and delineate the fundamental assumption it imposes on the network. Next, we detail this protocol and the typical structure of a program which uses it. Last, we discuss deadlock and overproduction issues involved in safely using this protocol.

2 Network Interface

The network interface hardware which Active Messages relies upon consists of two FIFOs which receive and send messages to and from the network and a set of status bits (see figure 1). The status bits represent the state of the messages in the send and receive FIFOs. Typically, the network interface is connected to the memory bus. Alternatively, the network interface may be connected to the I/O bus, the cache bus, or integrated with the processor registers as in the case of the J-Machine and Monsoon systems. Some network interfaces are associated with DMA engines which process them directly to memory.

Messages are sent and received through the network interface via a few simple steps. To send a message, the status bits for the send FIFO are checked to determine if it is free. Then the message is written into the send FIFO and dispatched. To receive a message, the status bits of the receive FIFO are checked to determine if a message has arrived. If so, the message is read from the receive FIFO, and the FIFO is subsequently cleared. To ensure proper transport of the messages, the Active Message protocol adheres to the following contract coined the *consumption assumption.*

> **Consumption Assumption:** *The network promises to deliver all messages eventually if processors promise to receive every message eventually.*

This assumption translates into the requirement that message sending routines cannot block. Otherwise, situations can arise (detailed later) to violate this assumption resulting in deadlock.

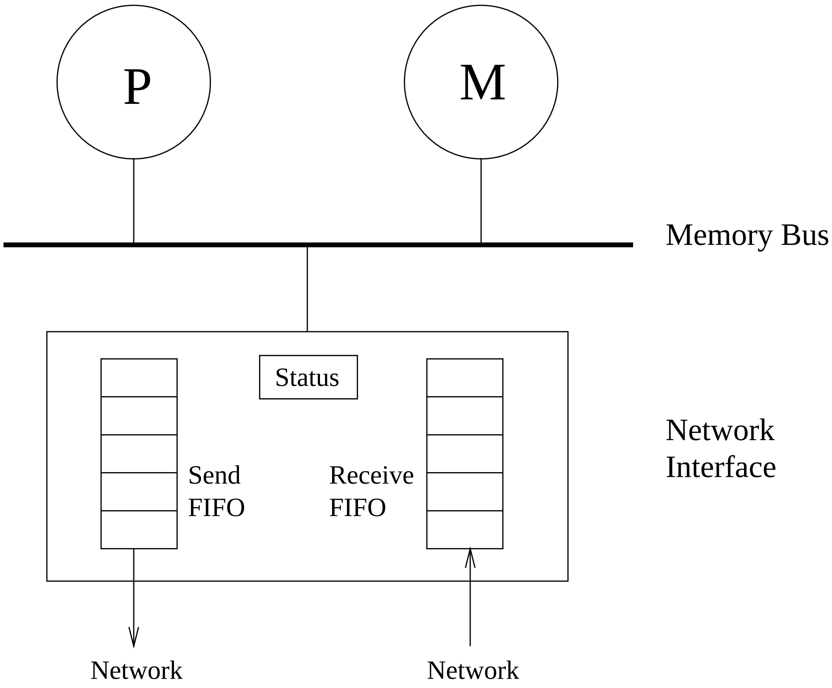

Figure 1: Network interface hardware.

3 Program Structure

Active Messages is a flexible, low-level protocol layer. Its message passing semantics are essentially restricted forms of remote procedure calls. A data packet is attached with a pointer to a handler, some remote procedure, which is invoked when the message arrives. The remote procedure's task is to quickly remove the message data from the receive FIFO and incorporate it into the computation with minimal work. The handler is run on top the user's stack. There is no separate space for handlers or buffering of messages. To facilitate the sender in specifying the address of the handler, programs are implemented within the context of the single program multiple data (SPMD) environment.

In the SPMD model, identical copies of a program run on all processors. Usually, the same memory map is utilized for both code and data to simplify references to each others' variables and handlers. However, in some implementations, the pointers to handlers are indexes to the addresses of remote functions. The following example in which a split-phase remote fetch is performed illustrates the structure of a typical parallel program implemented with Active Messages.

Example: y = processor 42's value of x.

Main function:

```
main() {
```

```
  if (self != 1) {
    while(1)
      poll();
  } else {

    arrived = FALSE;

    /*  (a)  (b)  (c)    (d) */
    rpc(42, &foo, self, &bar);

    /*  (a): destination processor id */
    /*  (b): request handler */
    /*  (c): caller's processor id */
    /*  (d): reply   handler */
    .
    .
    .
    while (!arrived) poll();
    ...
  }
}
```

Handlers:

```
foo(m, *reply_handler) {
  rpc(m,reply_handler,x);
}

bar(v) {
  y = v;
  arrived = TRUE;
}
```

Processor 1 sends a message to invoke the `foo()` handler on processor 42, which returns the value of x. Upon receipt of x at processor 1, the `bar()` handler is invoked to set the values of y and the *arrived* flag, indicating the transaction is complete. The execution of the above code is depicted in figure 2.

4 Sending and Receiving Messages

In a standard "active" message , the address of the remote procedure is prepended to the transported data as shown in figure 3.

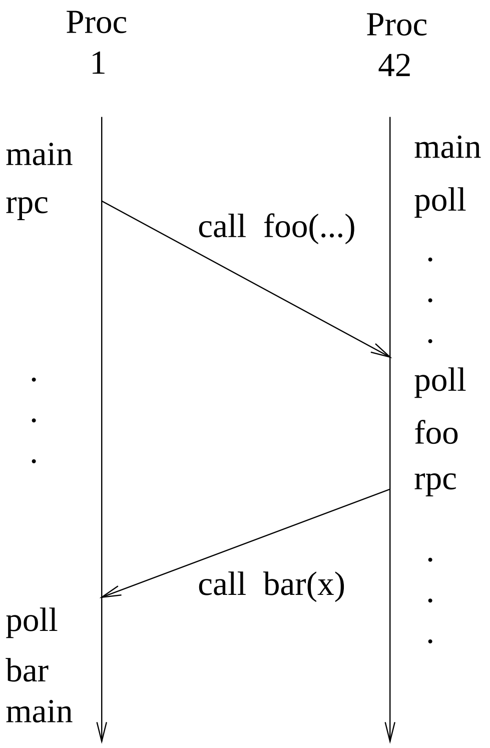

Figure 2: Execution of example program.

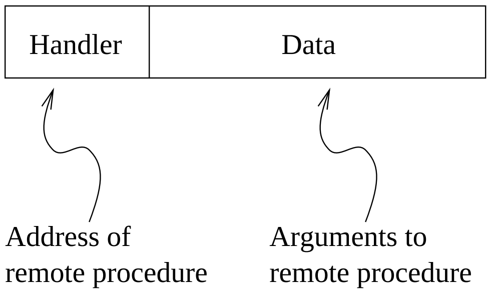

Figure 3: Standard message format.

Messages can be received in two different ways. The interrupt driven method for receiving messages is the least popular of the two. With this mechanism, as a message arrives, the current thread in the processor is interrupted, and the specified handler is invoked with the message data as arguments. This mechanism is slow and is prone to dangers with atomicity. Instead, explicit calls (`poll()`) to check for messages are usually inserted in programs. Known as polling, this convention is faster and avoids atomicity issues.

```
poll() {
  while (receive FIFO has message)
    call appropriate handlers;
}
```

A remote procedure call, `rpc(n, &foo, data)`, initiated by the sender invokes `foo(* data)` on processor n once the message arrives and processor n polls. From the sender's point of view, the send blocks until the message is sent (but not necessarily until received). This implies that the send implementation must poll to avoid violating the consumption assumption. Otherwise, two processors whose receive FIFOs are full might simultaneously send messages to one another. Because sending blocks, the processors would fall into dead lock. Pseudocode for the rpc implementation is given below. Once the remote procedure call returns, it is safe to manipulate `data`.

```
rpc(n, foo, data) {
  while (message not sent) {
    poll();
    try to send message (&foo, data);
  }
}
```

5 Protocol Issues

Similar to other low level message passing protocols, avoiding deadlock can be tricky business with Active Messages. The implicit polling in remote procedure calls or non-blocking sends ensure the protocol design is deadlock free; however, it opens the door for another danger known as *overproduction*. This danger arises as a result of remote procedure calls within a handler.

As mentioned before, the Active Messages protocol has no explicit buffering. However, the implicit poll in the rpc convention induces buffering via the user call stack. Consider the following handler `foo()`.

```
foo() {
  rpc(n, &bar, ...);
}
```

A processor when attempting to send a reply within the `foo()` handler might encounter another message which invokes `foo()` again. Thus, the call stack is incremented, implicitly buffering the previous message. If the processor is bombarded with messages invoking `foo()`, an indefinite recursion can occur possibly overflowing a finite stack.

```
call stack:  foo
             rpc (network busy)
             poll
             foo
             rpc (network busy)
             poll
             foo
             .
             .
             .
```

This recursion can be limited by limiting the number of messages sent to one processor. Some implementations leave this responsibility to the programmer. Others place a restriction on the send protocol by enforcing two types of remote procedure calls: request and reply. The request can only be sent from the main program, and a reply can only be sent within a request handler. Further, the reply handler cannot issue any type of send. Another simpler restriction would be to limit the number of requests.

HotBot is an example of a scalable system which utilizes Active Messages. It is an internet search engine implemented on a network of workstations communicating with fault-tolerant Active Messages. In this modified scheme, if the network is unable to deliver a message it returns the message to the handler within the caller.

References

[1] von Eiken, Thorsten, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. "Active Messages: a Mechanism for Integrated Communication and Computation." In *Proceedings of the 19th Annual International Symposium on Computer Architecture,* pages 256-266, Gold Coast, Australia, May 1992.

[2] HotBot FAQ: *The Technology Behind HotBot.* http://www.hotbot.com/FAQ/index.html?SEC=tech.