Generic Active Message Interface Specification

This Framemaker document lives in:
/disks/barad-dur/now/AM/spec/gam_spec.doc.
Please send any comments to cal@now.CS.Berkeley.EDU

The following people are to blame:
David Culler, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, Chad Yoshikawa

1 Introduction

Generic active messages support communication between the instances of a SPMD parallel program. This specification of them assumes that:

- The set of process instances in a parallel program are created and distributed across a set of processing nodes via some system service external to the message layer (e.g. glunix).
- Once running the instances are ranked and named with a virtual node number. Virtual node numbers are integers in the range of 0 to N - 1, where N is the total number process instances in the parallel program.
- This version of the interface provides a portability layer which will evolve over time. Its intention is to provide a messaging layer which will enable GLUNIX, library, and applications development for a prototype NOW.
- This version of the interface is not intended for multi-threaded programming and client/server computing. These issues will be addressed in future versions.

The generic active message layer contains five primitive operations: request, reply, store, get, and poll. Several utility operations are also provided for retrieving process configuration information such as the caller's virtual node number, the number of virtual nodes, and the maximum number of bytes supported by store and get.

2 Usage Paradigm

Active message communication is formulated as logically matching request and reply operations. Upon receipt of a request, a request handler is invoked; likewise, when a reply is received, the reply handler is invoked. Request handlers can reply at most once to the requesting node. If no explicit reply is made, the layer may generate one. Thus a request handler can call a reply at most once. It is an error for a request handler to reply to any node other than the requesting node. Some implementations will signal this error. Reply handlers cannot request or reply. The layer provides STORE and GET operations for block memory copies between nodes. A STORE operation is a request operation. A GET operation is a reply. (GET's can be thought of as an implicit request followed by a "STORE back" to the requesting node.)

2.1 Delivery Contract

The message layer will deliver requests and replies to destination virtual nodes barring any catastrophic errors (e.g. node crashes or msg layer bugs). From a sender's point of view, the request and reply functions block until the message is sent. A message is defined to be sent once it is safe for the caller to reuse the storage (registers or memory) containing the message. In implementations which copy or buffer messages for transmission, the definition still holds: message sent means the layer has copied the message and promises to delivery the copy with its "best effort", and the original message storage may be reused. By best effort, the message layer promises it will take care of all the details necessary to transmit the message. These details includes any retransmission attempts and buffering issues on unreliable networks. However, in either case, sent does not imply received. Once control returns from a request or reply function, users cannot assume that the message has been received and handled at the destination. The message layer only guarantees that if a request or reply is sent, and, if the receiver polls for arriving messages, then the message will eventually be received and handled. From a receiver's point of view, a message is defined to be received only once its handler function is invoked. The contents of partially received messages and messages whose handlers have not executed are undefined.

The message layer provides primitives for moving contiguous blocks of memory between two nodes. The STORE operation copies data to a remote memory region. Once the specified number of bytes have been written into the remote region, the request handler function is invoked on the destination node. The source memory region should not be modified until the STORE() function returns. The contents of the destination memory region is undefined until the request handler executes. The GET operation retrieves data from a remote (source) memory region. Once the specified number of bytes have been received into the local (destination) region, the reply handler function executes on the local node. The source memory region should not be modified during the lifetime of the GET operation. The destination memory region should not be modified until the reply handler function executes. The contents of the destination memory is undefined until the reply handler executes.

2.2 Data Types

For each active message function, an ANSI C prototype is specified along with a brief description of its operation, return values, and notes on its correct usage. The prototypes use abstract types for virtual node numbers and handler functions:

```
typedef unsigned int vnn_t;
```

```
typedef void (*handler_t)();
```

2.2.1 Function: am_enable()

```
void am_enable(...)
{
}
```

This call enables active messaging on the fast network. It should be called on every node before attempting any of the following operations. Implicitly, `am_enable()` acts as a barrier and performs a global synchronization among all participating instances of the parallel program. The arguments of `am_enable()` is dependent on the specific platform.

2.2.2 Function: am_procs()

```
int am_procs(void)
{
}
```

Returns total number of process instances comprising the parallel program. The value returned is one more than the largest virtual node number in the parallel program. Node numbers in parallel programs are ranked from 0 to `am_procs()` - 1.

2.2.3 Function: am_my_proc()

```
int am_my_proc(void)
{
}
```

Returns the virtual node number of the caller. The value returned is in the range of 0 to `am_procs()` - 1, inclusive.

2.2.4 Function: am_request_M()

```
int am_request_M(vnn_t dest,
                 handler_t handler,
                 int arg0, ..., int argM-1
{
}
```

This function blocks until the specified request message is sent. There are M=4 versions of the `am_request` function, each of which is tailored to pass between zero and four integer arguments to the remote request handler function.

The request handler is passed the requesting node's virtual node number, and between zero and four integer arguments. The number of arguments depends on the form of the request function. The handler can reply at most once to the requesting node using any one of the active message reply functions. The handler prototype is:

```
void handler(vnn_t requesting_vnn, arg0, ..., argM-1);
```

`am_request` must not be called from any handler function. It returns 0 on success, and -1 on error.

2.2.5 Function: am_request_df()

```
int am_request_df(vnn_t dest,
                  handler_t handler,
                  int arg0, int arg1,
                  double arg2)
{
}
```

Similar to the previous `am_request` functions, except that two integer values and one double-word floating point value may be passed to the request handler function. (The intent of this function is to allow the layer to optimize the movement of the double precision floating point value to the network interface.) Restrictions and returns values are the same as for `am_request`. The handler prototype is:

```
void handler(vnn_t requesting_vnn, arg0, arg1, arg2);
```

2.2.6 Function: am_reply_M()

```
int am_reply_M(vnn_t requesting_vnn,
               handler_t handler,
               int arg_0, ..., int arg_M-1)
{
}
```

This function blocks until the specific reply message is sent. There are M=4 versions of the `am_reply` function, each of which is tailored to pass between zero and four argument to the remote reply handler function.

The reply handler is passed the replying node's virtual node number, and between zero and four integer arguments. The number of arguments depends on the form of the reply function. The reply handler cannot call reply request or reply functions. The prototype of the reply handler is:

```
void handler(vnn_t replying_vnn, arg_0, ..., arg_M-1);
```

`am_reply` may only be called from a request handler function. It returns 0 on success, and -1 on error.

2.2.7 Function: am_reply_df()

```
int am_reply_df(vnn_t dest,
                  handler_t handler,
                  double arg0,
                  int arg1, int arg2)
{
}
```

Similar to the previous `am_reply` functions, except that two integer values and one double-word floating point value may be passed to the reply handler function. (The intent of this function is to allow the layer to optimize the movement of the double precision floating point value to the NI.) Restrictions and returns values are the same as for `am_reply`. The handler prototype is:

```
void handler(vnn_t replying_vnn, arg0, arg1, arg2);
```

2.2.8 Function: am_store()

```
int am_store(vnn_t dest_vnn,
            void *lva,
            void *rva,
            int nbytes,
            handler_t request_handler,
            void *handler_arg)
{
}
```

`am_store()` copies the caller's memory region named by (`lva`, `nbytes`) to the specified destination vnn's memory region named by (`rva`, `nbytes`). The source memory region should not be modified until after `am_store()` returns.

Upon receipt of the `nbtyes` of data, the receiver (`dest_vnn`) invokes the request handler function with the requesting node's virtual node number(`dest_vnn`), `handler_arg`, `rva`, and `nbytes` as arguments. The contents of the remote address space region (`rva`, `nbytes`) are undefined until the handler is invoked. The prototype of the request handler is:

```
void handler(vnn_t requesting_vnn, void *rva,
             int nbytes, void *handler_arg);
```

The maximum transfer size (`nbytes`) is given by `am_max_size()`. `lva` and `rva` may have arbitrary alignments, though implementation are expected to optimize for page, cache-line, and double-word alignment. Such optimizations may be performed at performance expense for word, short, and byte aligned cases.

This function returns 0 on success, and -1 on error. It may not be called from any handler function.

2.2.9 Function: am_store_async()

```
int am_store_async(vnn_t dest_vnn,
            void *lva,
            void *rva,
            int nbytes,
            handler_t request_handler,
            void *handler_arg,
            handler_t endfunc,
            void *endfunc_arg)
{
}
```

`am_store_async()` copies the caller's memory region named by (`lva, nbytes`) to the specified destination vnn's memory region named by (`rva`, `nbytes`) asynchronously. `am_store_async()` returns as soon as the memory transfer request is made. This allows computation to be overlapped with communication on the sender. When it is safe to reuse the source memory region, the callback function, `endfunc`, is invoked with the receiver's virtual node number (`dest_vnn`), `lva`, `nbytes`, and `endfunc_arg` as arguments. The source memory region should not be modified until `endfunc` is called. The prototype of the callback function is:

```
void endfunc(vnn_t dest_vnn, void *lva,
             int nbytes, void *endfunc_arg);
```

Upon receipt of the `nbytes` of data, the receiver (`dest_vnn`) invokes the request handler function with the requesting node's virtual node number(`dest_vnn`), `rva`, `nbytes`, and `handler_arg` as arguments. The contents of the remote address space region (`rva`, `nbytes`) are undefined until the handler is invoked. The prototype of the request handler is:

```
void handler(vnn_t requesting_vnn, void *rva,
             int nbytes, void *handler_arg);
```

The maximum transfer size (`nbytes`) is given by `am_max_size()`. `lva` and `rva` may have arbitrary alignments, though implementation are expected to optimize for page, cache-line, and double-word alignment. Such optimizations may be performed at performance expense for word, short, and byte aligned cases.

This function returns 0 on success, and -1 on error. It may not be called from any handler function.

2.2.10 Function: am_get()

```
int am_get(vnn_t source_vnn,
           void *rva,
           void *lva,
           int nbytes,
           handler_t reply_handler,
           void *handler_arg)
{
}
```

`am_get()` retrieves the source_vnn's memory region named by (`rva`, `nbytes`) to the caller's memory region named by (`lva`, `nbytes`). Upon receipt of the specified amount of data, the calling vnn invokes the reply handler with source node's virtual node number(`source_vnn`), `handler_arg`, `lva`, and `nbytes` as arguments. The contents of the caller's memory region named by (`lva`, `nbytes`) are said to be undefined until the handler function executes.The maximum transfer size (`nbytes`) is given by `am_max_size()`.The prototype of the reply handler is:

```
void handler(vnn_t replying_vnn, void *lva,
             int nbytes, void *handler_arg);
```

`lva` and `rva` may have arbitrary alignments, though implementation are expected to optimize for page, cache-line, and double-word alignment. Such optimizations may be performed at performance expense for word, short, and byte aligned cases.

This function returns 0 on success, and -1 on error. It may not be called from any handler function.

2.2.11 Function: am_max_size()

```
int am_max_size(void)
{
}
```

Returns maximum size (i.e. maximum nbytes parameter) for the store and get operations.

2.2.12 Function: am_disable()

```
int am_disable(void)
{
}
```

Call before program exits.

2.2.13 Function: am_poll()

```
void am_poll(void)
{
}
```

`am_poll()` will poll the network for all arriving messages. However, since messages may arrive while `am_poll()` is returning back to the caller, there may still be pending messages by the time `am_poll()` returns to the caller. `am_poll()` must be called in loops waiting for an event which requires a network operation.