Evolution of an InfiniBand API

By Jim Mott

VIEO founder Jim Mott examines the process his company followed to arrive at the interface used by InfiniBand managers and agents to send and receive Management Datagrams (MADs). Jim outlines the process from beginning to completion, providing data structures, functions, and a walk-through of two specific examples of the VIEO management API.

The InfiniBand architecture standard describes the functions that fabric elements must support to be managed or to provide management function to other elements. There are no definitions on how these functions should be implemented or what specific interface management applications should use to access a particular function.

VIEO provides prepackaged InfiniBand management and transport software for all defined fabric elements: target channel adapters, switches, host channel adapters, and routers. In order to provide this software, we must define and implement a number of programming interfaces. This article examines the process we followed to arrive at the interface used by InfiniBand managers and agents to send and receive MADs.

We began the process by looking at high-level requirements for this Application Program Interface (API). Who were the customers of the API? How many different environments must support the API? What other InfiniBand activities might be competing with management applications for access to the InfiniBand fabric? What sort of special boundary cases must be considered when evaluating potential APIs?

Finding the answers to these questions allowed VIEO to create a very high-level conceptual model for the environment in which InfiniBand management occurs. This environment is pictured in Figure 1.

Managers and agents will run in many different environments. The primary differentiators between environments are the InfiniBand chips that must be supported and the operating systems and run-time environments in which that code executes. These fundamental choices are represented in our conceptual model as the "ASIC Support Code" and "OS Abstraction Layer" boxes. Everything else is built on these two legs.

In addition to agents and managers, the VIEO framework must support other consumers of InfiniBand fabric services. These include interfaces to external enterprise and System Area Network (SAN) managers, InfiniBand-aware applications, and the standard device drivers that access information currently carried on the PCI bus. These functions are represented by the "Fabric Executive" and "O/S Bypass SDKs" boxes in the diagram.

The "VIEO Common Services Framework" box represents code that multiplexes between the different consumers of fabric services, provides functions that are common to different services, implements a security model that controls access to subnet services, and generally defines a robust execution environment for portable manager and agent code that is independent of specific chips or operating systems.

In order to assure the highest level of functionality and interoperability testing, we created portable library code that allows us to use exactly the same manager and agent code in each environment. This results in the majority of the development and test effort required to support a new environment being focused on the portability layer. Managers, agents, and transports can be debugged in the environment with the best tools and regression can be tested in new ones.

The VIEO Management Application Interface (MAI) library implements the primary transport between InfiniBand managers/agents and the fabric. This API

Figure 1

Copyright 2001 CompactPCI Systems

Reprinted from CompactPCI Systems / September 2001

provides the functionality described in the InfiniBand specification as Subnet Manager Interface (SMI) and General Services Interface (GSI). In addition, it provides raw access to the fabric for debuggers and extended protocol development. It also supports migration and takeover of the manager/agent function within a single environment.

MAI provides an interface for agents and managers to select exactly which Subnet Management Packets (SMPs) and General Services Management Packets (GMPs) they wish to receive based on physical source (channel, port), MAD header fields (Global Route Header, Local Route Header, etc.), base MAD fields (version, class, method), and quantifiers (offset, length, value) into the class specific fields. The interface also supports migration of agent and manager function during the boot process, the development process, and end customer code deployment and update processes.

The MAI library provides the only access method to Queue Pair 0 and Queue Pair 1 for agents and managers. The management API is supported at all levels of the VIEO software stack which means that agents and managers can run at any level of that stack. It also means that agents and managers can coexist with other agents and managers running at the same or different levels within the software stack.

The API supports the concept of consuming or non-consuming agents and managers. When a piece of code registers a filter that describes the SMPs or GMPs it wishes to receive, it also requests that MADs that pass this filter are either delivered exclusively to this function or delivered to all agents or managers with filters that would pass the MAD.

Access to redirected General Services Interface ports is outside the scope of the management API. If a manager or agent redirects a general service management class, then that manager or agent is responsible for setting up and servicing all I/O for the redirected Queue Pair.

All data is exchanged between the MAI library calls and management applications using two data structures. These structures are:

- mai_mad_t – Holds a complete MAD plus routing information
- Filter_t – Holds a MAD template using select MADs to receive

The MAI library provides support for the following functions:

- mai_open – Open a channel to QP0 or QP1 on a specific device + port
- mai_close – Close an open channel and release all resources
- mai_filter_create – Describe MADs to be received on an open channel
- mai_filter_delete – Remove a filter
- mai_send – Send a MAD on an open channel
- mai_recv – Receive the next MAD matching an attached filter

These are the only necessary functions that a manager or agent uses to send and receive MADs from QP0 or QP1 on any available InfiniBand device. The API also supports access to all ports on multi-port channel adapters (HCAs and TCAs) and routers.

The MAI library provides support for endian independent data conversions of the core data structures used in the system. These data conversion functions are:

- mai_mmstream() – Convert mai_mad_t to byte stream
- mai_rmstream() – Inverse of mai_mmstream()
- mai_mfstream() – Convert Filter_t to byte stream
- mai_rfstream() – Inverse of mai_mfstream()
- mai_towire() – Create InfiniBand MAD from mai_mad_t
- mai_fromwire() – Create mai_mad_t from an InfiniBand MAD

The MAI library is implemented using a few of the operating system abstraction functions from the "OS Abstraction Layer" box on Figure 1. To convey an idea of what sort of functions are provided by this layer, the functions and a short description are shown below:

- vs_initlock – Initialize a spin lock
- vs_lock – Get a spin lock
- vs_unlock – Release a spin lock
- vs_thread_create – Create a thread, if possible
- vs_thread_resume – Begin execution of a thread
- vs_thread_kill – Kill a thread (me in this file)
- vs_thread_name – Returns pointer to thread CB
- vs_event_create – Initialize and event ctl blk
- vs_event_wait – Wait for an event
- vs_event_post – Signal an event has occurred
- vs_time_get – Return 64-bit current time in microseconds
- vs_log_error – Store fatal error log
- vs_log_info – Interesting trace info
- vs_enter – Function entry
- vs_exit – Function exit
- vs_fatal_error – Shutdown total system
- vs_priv_check – Validates appropriate privileges

In addition to the operating system services shown above, a number of environment-specific functions (user space, kernel space, embedded, etc.) from the "OS Abstraction Layer" are also used in the implementation of the MAI interface. These functions are:

- ib_attach_sma – Attach QP0/QP1 down channel
- ib_detach_sma – Let it go again
- ib_recv_sma – Read one MAD from down channel
- ib_send_sma – Send a MAD to down channel
- ib_control – Manipulate down channel filters

To better illustrate how the VIEO management API works, we will walk through two specific examples. These examples assume a full-blown HCA environment with a user space agent, kernel support, and an intelligent InfiniBand adapter.

The important point here is that the example agent code and the management API that supports it, run totally unmodified in all the possible locations in the software stack. This example shows a three-level stack: user space, kernel space, embedded firmware. The agent is shown running in user space, but it runs equally well in the kernel or embedded environment.

Copyright 2001 CompactPCI Systems

Typically a TCA or a switch will truncate the top one or two levels of the stack and run the agent on the highest remaining level. Many HCAs will not implement the intelligent adapter, so they will truncate the bottom level of the stack. None of these differences will have any effect on the agent source code or on the correct operation of the agent.

Figure 2 shows the management flow through the software framework.

mai_open flow

Before the agent can send or receive MADs, it must establish a connection through the management API to the InfiniBand chipset that actually puts MADs on the wire and receives them from the wire. The basic flow for this open operation looks as follows:

1. **Agent calls mai_open().US**
   The user space mai_open() function notices that it does not have an open connection to the management API open service below it. It uses the common services function ib_attach_sma() to open a channel to the MAD send/receive subsystem.

   Once the ib_attach_sma().US call returns without error, the library code starts a thread that immediately calls ib_recv_sma() on the newly opened channel to the management API service below it. As this thread receives MADs, it matches them against filters attached to all the up channels and places the MAD in the receive queue for each channel that has a matching filter.

   Note: There are more details to the receive, filter matching, and MAD queuing operations. There is also a non-threaded version of the management API library that does not require a receive thread or any other thread support. This implementation is not discussed in detail.

2. **ib_attach_sma().US uses ioctl()**
   The user space common services implementation of ib_attach_sma() uses an ioctl() call to connect with the kernel management API implementation.

3. **In kernel device driver ioctl() handler calls mai_open().K**
   The ioctl() handler in the kernel device driver runs in the callers process context and requests the management API to open a channel. Notice that this has been placed in kernel agents/managers at the same level as user space agents/managers as far as the in kernel management API is concerned.

4. **mai_open().K calls ib_attach_sma().K**
   The kernel space mai_open() function notices that it does not have an open connection to the MAD management API open service below it. It uses the common services function ib_attach_sma() to open a channel to the MAD send/receive subsystem.

   Once the ib_attach_sma().K call returns without error, the library code starts a thread that immediately calls ib_recv_sma() on the newly opened channel to management API service below it. As this thread receives MADs, it matches them against filters attached to all the up channels and places the MAD in the receive queue for each channel that has a matching filter.

5. **ib_attach_sma().K send message to firmware**
   The in kernel implementation of ib_attach_sma() packages up the open request and sends it to the embedded processor on the intelligent HCA adapter to service. The IPC used to communicate between the kernel and the embedded processor is part of the kernel porting work required to support the HCA.

6. **Embedded HCA processor extracts message and calls mai_open().FW**
   Communication between the host and the embedded processor runs through a messaging library. When a request from the host is

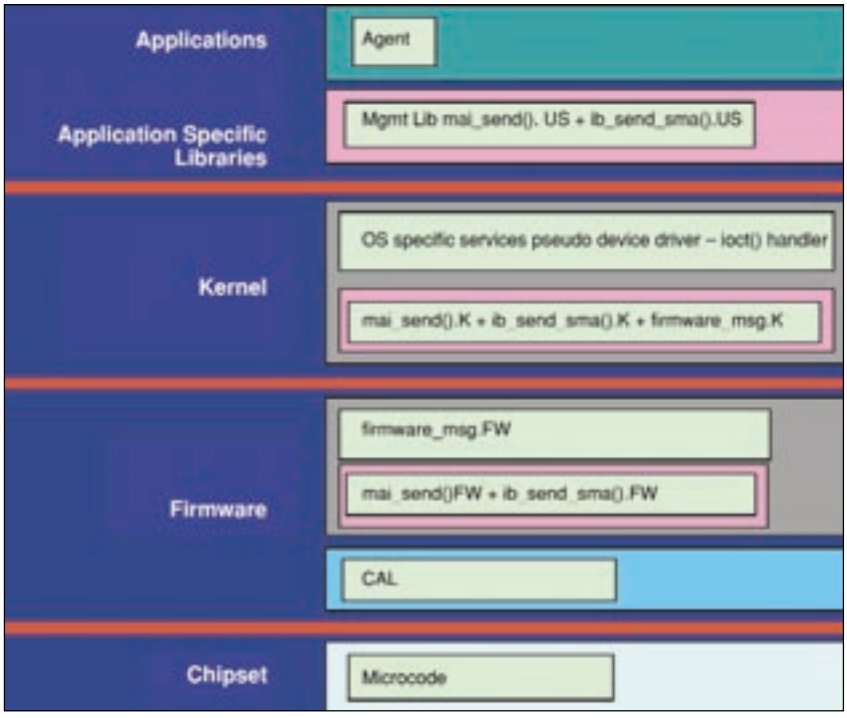

Figure 2

Copyright 2001 CompactPCI Systems

received, the function requested is decoded by the messaging subsystem and the appropriate entry point is called. Notice that this has placed all the agents/manager running in the host at the same level as firmware agents/managers, as far as the firmware management API is concerned.

7. **mai_open().FW calls ib_attach_sma().FW**
   The embedded mai_open() function notices that it does not have an open connection to the MAD management API open service below it. It uses the common services function ib_attach_sma() to open a channel to the MAD send/receive subsystem.

   Once the ib_attach_sma().FW call returns without error, the library code starts a thread that immediately calls ib_recv_sma() on the newly opened channel to management API service below it. As this thread receives MADs, it matches them against filters attached to all the up channels and places the MAD in the receive queue for each channel that has a matching filter.

8. **ib_attach_sma().FW interacts with InfiniBand chip to start MADs flowing**
   The firmware version of ib_attach_sma() is responsible for actually working with the InfiniBand chipset to make sure management MADs can be sent and received on the SMI or GSI interface. It does this by using "ASIC Support Code" functions, internal serialization functions, and any other resources required.

In this example flow, three different types of code were required:

- **Common code**
  The management API library (mai_open) is exactly the same code running in three different environments: user space, kernel space, and firmware.

- **OS Abstraction Layer transport code**
  The common services porting layer function (ib_attach_sma) is implemented differently in each of the three different environments. This is the primary work required to support a new layer in the stack.

- **Glue code**
  The various environments require different mechanisms to exchange information across their boundaries (ioctl(), messages, CSR I/O). While this code must be developed for each port, it can be shared between different communicating entities in one port and across different but similar ports to different platforms.

mai_send flow

Once a management channel has been opened, applications are able to send MADs out that channel. These MADs can be fully specified (raw MADs), including all headers, optional headers, and data fields or they can be "normal" MADs. The management API library code is responsible for verifying that applications sending raw MADs have appropriate privileges to do that.
The basic flow for an mai_send operation looks like:

1. **Agent calls mai_send().US**
   The user space mai_send() function validates the request and passes the mai_mad_t data structure to ib_send_sma().US.

2. **ib_send_sma().US uses ioctl()**
   The user space common services implementation of ib_send_sma() uses an ioctl() to pass the MAD to kernel space management API code.

3. **In kernel device driver ioctl() handler calls mai_send().K**
   The device driver ioctl() handler acts as a kernel space surrogate for the management API library code running in user space processes. It runs on the user space process/thread and calls the in kernel mai_send().K function.

4. **mai_send().K calls ib_send_sma().K**
   The kernel space mai_send() function validates the request and passes the mai_mad_t data structure to ib_send_sma().K.

5. **ib_send_sma().K sends a message to firmware**
   The kernel implementation of ib_send_sma() packages up the request and the mai_mad_t data and sends it to the embedded processor on the HCA adapter. The IPC used to communicate between the kernel and the embedded processor is part of the kernel porting work required to support an HCA. This is the same IPC code used by other kernel space ib_xxx_sma() functions.

6. **Embedded processor on HCA adapter extracts message and calls mai_send().FW**
   When the adapter on the firmware receives a message from the kernel, it decodes the message and data. It then calls mai_send().FW with the mai_mad_t data.

7. **mai_send().FW calls ib_send_sma().FW**
   The firmware mai_send() function validates the request and passes the mai_mad_t data structure to ib_send_sma().FW.

8. **ib_send_sma().FW interacts with the InfiniBand chip to send out the MAD**
   The firmware implementation of ib_send_sma() is responsible for doing whatever hardware functions it takes to put the MAD out on the wire. It does this by using "ASIC Support Code" functions, internal subroutines and serialization functions, and any other resources required.

   The lowest level ib_send_sma() function is also responsible for special processing on Out Of Band (OOB) messages. When the mai_mad_t contains no MAD, but instead, OOB data, the ib_send_sma().FW routine will wrap the mai_mad_t structure back to the received MAD queue. This results in the OOB data being passed back up the stack and queued to the input buffers of all channels with matching filters. This function provides a way for agents/managers that are notified of another agent/manager taking over exclusively what used to be their

MADs, to provide the new agent/ manager with current internal state.

In this example flow, four different types of code were required:

- **Common code**
  The management API library (mai_send) is exactly the same code running in three different environments: user space, kernel space, and firmware.

- **OS Abstraction Layer transport code**
  The common service porting layer function (ib_send_sma) is implemented differently in each of the three different environments. This is the primary work required to support a new layer in the stack.

- **Endian independent data conversion functions**
  When the ib_send_sma().K function passes the mai_mad_t data structure to firmware it must first render that structure as a byte stream that preserves structure element boundaries and interpretations. It does this by calling the mai_mmstream() utility function. The firmware message handler code uses the inverse mai_rmstream() utility to convert that byte stream to a firmware version of the mai_mad_t structure suitable for passing to mai_send().FW.

- **Glue code**
  The various environments require different mechanisms to exchange information across their boundaries (ioctl(), messages, CSR I/O). While this code must be developed for each port, it can be shared between different communicating entities in one port and across different but similar ports to different platforms.

Summary

The publication of the InfiniBand Architecture 1.0 Specification in October 2000 was a culmination of a tremendous collaborative effort by the leading technology companies heading up the InfiniBand Trade Association (IBTA). As the industry gears up to deliver commercial InfiniBand-enabled products, the active working groups within the IBTA continue to expand the specification to ensure it addresses the functional needs of tomorrow's data center. However, the specification deliberately avoids defining how management functions should be implemented, leaving that to the vendors deploying InfiniBand products.

This article outlining the VIEO MAI is contributed to the InfiniBand community in an effort to promote dialog and to accelerate industry adoption. It is the first in a series of papers that describe VIEO's software architecture and APIs. The author welcomes comments and the opportunity to discuss specific implementation issues.

***Jim Mott** is the Chief Technology Officer and founder of VIEO, Inc. He has more than 20 years experience in computer system design, architecture, and implementation. With both hardware and software expertise, Jim has an extensive background in communications architecture, networking, cluster distributed systems, embedded systems, video servers, robotics, and instruction and gate-level computer simulators. He also served as lead architect on IBM's AIX operating system.*

For more information contact Jim at:

James M. Mott
VIEO, Inc.
12416 Hymeadow Dr., Suite 200
Austin, TX 78750
Telephone: 512-257-3031
E-mail: jim@vieo.com
Web site: www.vieo.com

Copyright 2001 CompactPCI Systems